Contributions à la vérification automatique de protocoles de groupes.

Najah Chridi

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Contributions à la vérification automatique de protocoles de groupes

THÈSE

présentée et soutenue publiquement le 11 Septembre 2009

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Najah CHRIDI

Composition du jury

Rapporteurs :	Maryline MAKNAVICIUS-LAURENT	Professeur, Telecom & Management, SudParis
	Ralf TREINEN	Professeur, Université Paris Diderot
Examinateurs :	Sjouke MAUW	Professeur, Université du Luxembourg
	Refik MOLVA	Professeur, Eurecom Sophia-Antipolis
	Isabelle CHRISMENT	Professeur, UHP Nancy 1
	Laurent VIGNERON (directeur)	Maître de conférences, Université Nancy 2
	Michael RUSINOWITCH (directeur)	Directeur de Recherche, INRIA
	Mathieu TURUANI	Chargé de Recherche, INRIA

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Table des matières

Table des figures

Liste des tableaux

1

Introduction

Sommaire

Les protocoles cryptographiques sont cruciaux pour sécuriser les transactions électroniques. Ce sont des règles d'échange entre les points du réseau pour assurer la sécurité des communications. Ils se basent sur des fonctions cryptographiques afin d'assurer des propriétés de sécurité telles que le secret ou l'authentification. La présence des protocoles cryptographiques dans des applications telles que le commerce électronique, la téléphonie mobile ou la télévision à la demande, fait de leur conception et leur sécurité un but important de recherche. Les protocoles cryptographiques devraient être vérifiés avant leur commercialisation car la moindre faille sur ces protocoles peut avoir des répercussions économiques graves. Malheureusement, vérifier la sécurité de ces protocoles est difficile. D'ailleurs, beaucoup se sont avérés défectueux après plusieurs années de mise en service (e.g. [75]). Dans les années passées, la vérification des protocoles cryptographiques standards gérant deux ou trois participants [34] a donné beaucoup de résultats intéressants dans le cadre du modèle de Dolev Yao [45]. D'intenses travaux de recherche sur ce modèle ont conduit au développement de plusieurs formalismes et outils automatiques de vérification des protocoles cryptographiques [4, 14, 16].

Avec le succès de la vérification de protocoles relativement classiques, des travaux récents s'attaquent à de nouvelles classes de protocoles, beaucoup plus complexes. L'une de ces classes comprend les protocoles de sécurité paramétrés, i.e. les protocoles dont la spécification dépend d'un ou plusieurs paramètres. Comme exemple, nous pouvons citer ceux utilisant des structures

de données non bornées telles que les listes. D'autres protocoles admettent comme paramètre le nombre de participants impliqués. Ces protocoles sont aussi appelés protocoles de groupe [65, 94, 95]. Par exemple, les protocoles de gestion de clefs construisent une clef commune entre un ensemble de participants afin de sécuriser leur communication.

La vérification de tels protocoles est confrontée à plusieurs problèmes. En effet, la sécurité des communications au sein de groupes n'est pas nécessairement une extension simple d'une communication sécurisée entre deux parties : le groupe peut changer de structure en ajoutant ou en supprimant un ou plusieurs membres. Ainsi, les protocoles de groupe nécessitent des propriétés de sécurité plus développées, liées à la dynamique du groupe. Par exemple, il faut vérifier que, dans un protocole d'accord de clefs, tous les membres arrivent à déduire la même clef de groupe. D'où le besoin d'étendre les outils et les méthodes existants dont la plupart se concentrent seulement sur les propriétés de sécurité classiques telles que l'authentification ou le secret. En outre, les protocoles de groupe fonctionnent avec un nombre non borné de participants. Cependant, la plupart des approches automatisées de vérification de protocoles nécessitent un modèle instancié : la taille du groupe doit être fixée à l'avance. Or, cette contrainte restreint la valeur d'une validation : si le protocole est sûr pour n participants, l'est-il aussi pour $n + 1$?

Ces difficultés ont fait que, peu de travaux existent pour la vérification des protocoles de groupe. Quelques-uns [57, 59, 60, 99] ont fourni des résultats de décidabilité très restrictifs. D'autres [73, 81, 94, 96] se sont contentés d'effectuer des travaux expérimentaux sur ces protocoles afin de détecter des attaques éventuelles.

Nous présentons dans ce qui suit les notions sur lesquelles se basent les protocoles cryptographiques. Le reste de ce chapitre sera consacré aux différents formalismes et méthodes utilisés pour la vérification de ces protocoles. Finalement, nous décrivons les différentes parties qui constituent ce manuscrit.

1.1 Protocoles cryptographiques

1.1.1 Protocoles de communication

Un protocole de communication peut être vu comme un échange de messages entre plusieurs entités. Ces entités sont définies par un certain nombre d'actions qu'ils doivent effectuer lors d'une exécution normale du protocole, i.e. sans intervention de personne malhonnête. Ces entités sont appelées participants, agents, principaux ou parties. Un protocole est donc une séquence de règles. Pour chacune de ses règles, trois informations sont données : l'émetteur, le récepteur et le message envoyé. Très souvent, on a besoin de considérer plusieurs instances ou déroulements du même protocole cryptographique. On appelle chaque instance une **session**. Ces différentes sessions peuvent être exécutées en parallèle, i.e. les règles de ces sessions peuvent être entrelacées. Pour différencier les sessions, on utilise généralement des nombres aléatoires de grandes tailles, générés par les participants, appelés **nonces**. Chaque protocole est conçu pour assurer un certain nombre d'objectifs. Nous pouvons donc définir un protocole cryptographique [90] comme étant une séquence d'échanges de messages entre des agents légitimes, qui a pour but d'offrir un ou plusieurs services de sécurité à ces mêmes entités.

Comme cité en introduction de ce chapitre, les protocoles cryptographiques sont omniprésents. Nous les utilisons inconsciemment tous les jours. Par exemple, pour le paiement par carte bancaire, nous utilisons un protocole cryptographique dont la version simplifiée est illustrée en Figure 1.1. Nous nous intéressons au protocole d'authentification locale, i.e. utilisé pour les achats d'un montant inférieur à une certaine somme (généralement 100 euros) et qui ne nécessite pas

FIG. 1.1 – Échange de messages pour le paiement bancaire

l'intervention de la banque fournissant la carte bancaire. Ce protocole implique trois entités : un acheteur, une carte bancaire et un terminal. La donnée *Data* contient le nom, le prénom, le numéro de carte et sa date de validité. La donnée VS est la signature de *Data* permettant d'assurer que cette donnée *Data* n'a pas été modifiée. Cette donnée ainsi qu'un code à quatre chiffres et une valeur de signature VS sont présents sur la carte. La valeur de la signature est calculée une fois pour toute par la banque et enregistrée sur la carte lors de son initialisation. En outre, la carte n'a aucun moyen de générer cette valeur de la signature. Le protocole fonctionne comme suit :

- Quand l'acheteur introduit sa carte dans le terminal, celui-ci authentifie la carte en lisant les deux données *Data* et VS de la carte.
- Une fois que cette authentification est réussie assurant qu'il s'agit bien d'une bonne carte, le terminal affiche le message *code*? pour inviter l'acheteur à taper son code. Après que celui-ci tape son code, ce dernier est transmis du terminal à la carte. La carte vérifie si le code est valide et le signale au terminal si c'est le cas.

Il existe beaucoup de protocoles cryptographiques dans la littérature. Une liste de ces protocoles peut être trouvée dans [34].

1.1.2 Protocoles de groupe

Au cours de la dernière décennie, nous avons assisté au développement d'applications coopératives impliquant un groupe d'utilisateurs, telles que l'enseignement à distance, les jeux de groupe interactifs ou la vidéo conférence. On parle alors de communications de groupe. Une communication de groupe implique un style de communication de plusieurs à plusieurs dans un groupe. Pour optimiser le temps d'acheminement de données, cette communication se fait généralement de un à plusieurs. Il s'agit alors d'une communication multi-points (*multicast* [42]). Pour la plupart des applications citées ci-dessus, les utilisateurs du groupe considéré doivent être souvent connectés rapidement sans infrastructure préalable, vu que ce groupe n'est pas permanent et que le déploiement d'une infrastructure est contraignant de point de vue coût et temps d'utilisation. En outre, ce groupe d'utilisateurs est souvent dynamique : les utilisateurs se déplacent d'une façon libre et arbitraire en quittant ou en joignant le groupe à tout moment. Les réseaux connectant ce type d'utilisateurs sont appelés réseaux mobile ad hoc (MANET), i.e. des réseaux sans infrastructure, ayant une topologie très dynamique. Ces réseaux sont surtout présents dans le domaine militaire ou encore pour la coordination des forces civiles.

Le recours aux communications multicast dans les réseaux ad hoc admet plusieurs avantages tels que le passage à l'échelle ou la flexibilité. Cependant, cette combinaison entraîne aussi des problèmes de sécurité. En effet, l'absence des identités des participants et la publication de l'adresse du groupe multicast peuvent causer des attaques de déni de services. En outre, l'ab-

sence d'infrastructure pour les réseaux ad hoc rend les noeuds vulnérables.

Pour assurer la sécurité de ces groupes d'utilisateurs, on a généralement recours aux protocoles de groupe. D'une manière générale, un protocole de groupe est un protocole par lequel, les participants du groupe échangent d'une manière sécurisée des informations afin d'aboutir à un but commun. Comme but possible, nous avons la signature d'un contrat (e.g. [48]) ou la déduction d'une clef commune (e.g. [7]).

La plupart des applications nécessitent le partage d'une clef commune entre les membres d'un groupe. Il existe différentes approches pour générer cette clef. Quelques-unes (e.g. [91, 105]) optent pour l'utilisation d'un serveur pour fournir cette clef au groupe. Il s'agit de l'approche centralisée. D'autres (e.g. [6, 56, 95]), appelées protocoles d'accord de clefs, exigent la contribution de chacun des membres du groupe à la construction de la clef. D'autres (e.g. [71]) utilisent des contrôleurs de sous-groupes, qui génèrent une clef entre eux et la diffusent, chacun, au sous-groupe qu'il contrôle.

Souvent, les applications de groupe nécessitent un traitement efficace des groupes dynamiques, i.e. les groupes où les agents adhèrent et quittent le groupe fréquemment, et où les groupes peuvent fusionner ou se séparer. Pour assurer les changements dynamiques des groupes, un protocole de groupe peut être défini par plusieurs sous-protocoles pour des opérations telles que le changement de la composition du groupe (e.g. ajout ou sortie d'un ou de plusieurs membres), ou le changement de la structure du groupe, présent surtout pour les protocoles hiérarchiques [86] (e.g. montée ou descente de classe). Des architectures de protocoles ont été alors proposées pour la génération des clefs de groupes et leur mis à jour par le biais de protocoles dédiés aux opérations de groupe. Comme exemple, nous citons le projet Cliques [7, 8, 95].

En résumé, les protocoles de groupe se caractérisent par le nombre arbitraire des participants et les propriétés spécifiques qu'ils doivent satisfaire, et qui sont dues à la dynamique du groupe. Par exemple, les protocoles de gestion de clefs sont des protocoles de groupe qui permettent à un groupe de participants de générer une clef commune et de la mettre à jour en fonction de l'évolution de la structure du groupe. La Figure 1.2 illustre un exemple de ces protocoles. Elle décrit un échange de messages entre n clients et un serveur afin de générer une clef commune pour sécuriser leurs futures communications. Chaque client i génère une contribution $Contrib_i$ et une clef $Clef_i$ et les envoie au serveur, cryptées par la clef publique de ce dernier : $Clef_S$. Une fois que le serveur construit la clef du groupe en utilisant toutes les contributions qu'il a reçues, il l'envoie à chacun des clients en le cryptant par la clef correspondante au client en question. Finalement, les clients et le serveur possèdent la même clef $Clef_G$ qui leur permet d'entamer une communication sécurisée. Le but de ce protocole est donc de garantir l'accord de clefs, i.e. tous les participants se mettent d'accord sur une seule valeur de la clef.

Les protocoles de groupe, tel que celui de la Figure 1.2, constituent un défi majeur pour la vérification des protocoles cryptographiques pour différentes raisons. D'abord, comme ces protocoles impliquent un nombre de participants non borné, l'ensemble de messages échangés est donc infini et même leur modélisation nécessite plus d'effort pour représenter de manière générique les échanges de messages. Comme exemple de cette situation, nous citons la phase d'envoi des contributions de la Figure 1.2.

Ensuite, lorsque le groupe comprend une entité centrale tel que le serveur, cette entité a besoin d'effectuer des traitements itératifs afin d'extraire des informations du flux de données qu'elle reçoit des autres participants. Par exemple, dans la Figure 1.2, le serveur a besoin d'extraire les contributions des participants pour construire la clef du groupe. Il a aussi besoin

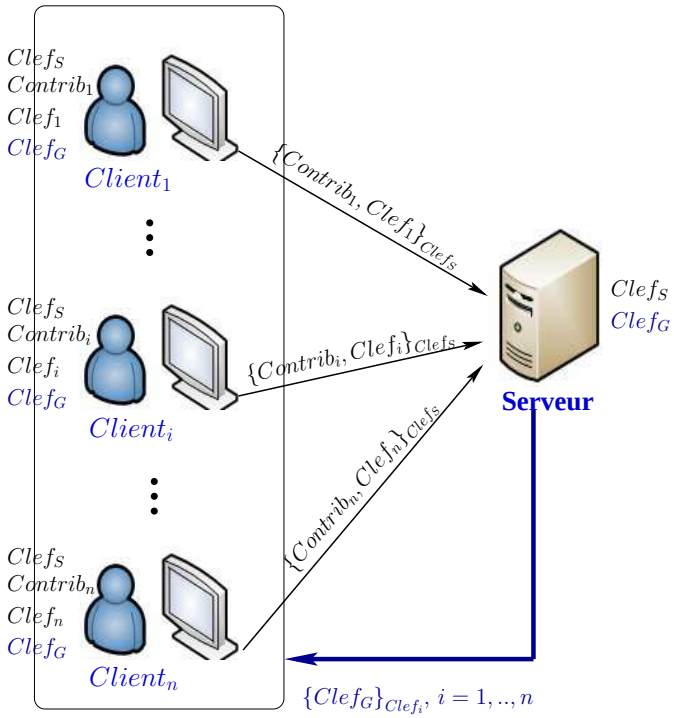


FIG. 1.2 – Exemple de génération d'une clef de groupe

d'extraire les clefs individuelles de ces participants pour l'envoi des messages à la fin.

Enfin, mis à part la propriété classique de secret, les protocoles de groupe exigent d'autres propriétés spécifiques. Par exemple, dans le cas du protocole de la Figure 1.2, la propriété à satisfaire est celle de l'accord de clefs.

1.1.3 Primitives cryptographiques

Pour composer leurs messages, les participants d'un protocole cryptographique ont souvent recours à des outils, appelés aussi primitives cryptographiques. Nous nous intéressons à trois types de ses primitives.

Concaténation

La première primitive cryptographique couramment utilisée est la concaténation. Cet opérateur est noté $\langle .., .. \rangle$. Une concaténation de trois messages u, v, w s'écrit sous la forme $\langle u, \langle v, w \rangle \rangle$.

Chiffrement

Une des principales primitives cryptographiques est le chiffrement (nommée aussi encryption) de données (voir Figure 1.3). Le chiffrement est l'action de transformer un message m en un message encrypté en utilisant une clef k, noté $\{m\}_k$. Le déchiffrement est l'opération inverse qui permet d'appliquer un algorithme de déchiffrement au texte chiffré $\{m\}_k$ en utilisant la clef adéquate, appelée clef de déchiffrement, afin d'obtenir le texte d'origine m.

Deux types de chiffrements sont à distinguer : le **chiffrement symétrique** et le **chiffrement asymétrique**. Le chiffrement symétrique utilise la même clef pour chiffrer comme pour déchiffrer. Le chiffrement symétrique d'un message m par une clef k est souvent noté $\{m\}^s_k$. Le

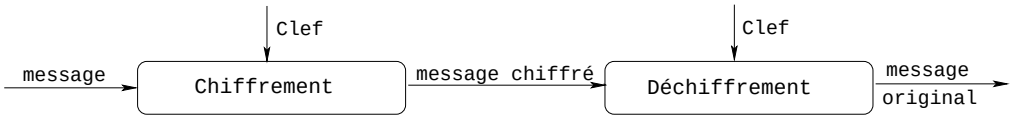


FIG. 1.3 – Chiffrement et déchiffrement

chiffrement asymétrique (voir figure 1.4) requiert deux types de clefs. La première est utilisée pour chiffrer et elle est souvent publique afin que tout participant puisse chiffrer. La deuxième clef est en général secrète pour qu'un seul puisse déchiffrer. Ce chiffrement d'un message m par une clef k est souvent notée $\{m\}_k^p$. L'un des algorithmes de chiffrement à clef publique les plus connus est l'algorithme RSA, dû à R.L. Rivest, A. Shamir et L.M Adleman [88].

FIG. 1.4 – Chiffrement asymétrique

Dans le cadre de la vérification des protocoles cryptographiques, une hypothèse très classique retenue est celle de **chiffrement parfait** [45]. Cette hypothèse assure en particulier, qu'il est impossible de décrypter un message chiffré sans avoir la clef correspondant à l'algorithme de déchiffrement. Cette hypothèse permet d'automatiser la vérification des protocoles. Notons aussi que la **signature** peut être codée comme un chiffrement asymétrique (et donc sous forme de $\{m\}_k$), avec la différence que la clef de chiffrement k est privée et la clef de déchiffrement est publique. Un participant qui reçoit un tel message $\{m\}_k$ peut vérifier la signature de celui qui a composé ce message, par le simple fait qu'il utilise comme clef de déchiffrement, la clef publique de ce prétendant.

Hachage

Une fonction de hachage permet d'associer à un message de longueur variable, un message de petite taille fixée. Les fonctions de hachage sont des fonctions à sens unique (*one-way functions*), i.e. la connaissance du message haché et la fonction de hachage h ne permet pas de déduire le message d'origine. Les fonctions de hachage sont souvent utilisées pour assurer l'intégrité d'un message, i.e. la non altération de ce message. En effet, étant donné un message m et son message haché $h(m)$, un participant peut s'assurer de l'intégrité du message m en le hachant par la fonction h et en comparant le résultat avec le message $h(m)$.

1.1.4 Propriétés de sécurité

Chaque protocole cryptographique a été conçu pour garantir certaines propriétés de sécurité. Les propriétés les plus fréquentes sont le secret et l'authentification. Les propriétés exigées d'un protocole dépendent très souvent du contexte d'utilisation du protocole. Par exemple, pour les protocoles de signature de contrats, les propriétés les plus utiles sont la non répudiation et l'équité. Pour les protocoles de vote électronique, la propriété d'anonymat est la plus demandée.

Concernant les protocoles de groupe, les propriétés d'accord de clef et de secret sont les plus utiles.

Secret La propriété de secret standard ou simple [45], souvent requise pour les protocoles cryptographiques, exprime le fait qu'un certain message ne doit être connu que par certains membres. En d'autres termes, une personne malhonnête ne doit pas être capable de deviner la valeur d'une expression qui correspond au secret. La manière la plus simple de sécuriser un secret est de le chiffrer par une clef qui n'est pas connue par l'intrus.

La plupart des méthodes de vérification de protocoles de groupe utilise cette notion de secret. Cependant, il existe une notion plus forte : le secret fort [13]. Cette propriété est formalisée par une équivalence observationnelle : étant donné deux versions du protocole où le secret est remplacé par deux messages différents, un pour chaque version, le secret est préservé si l'intrus ne peut pas distinguer entre ces deux versions.

Des variantes de la propriété de secret comme le secret futur expriment généralement le fait que la révélation d'informations à l'intrus (généralement des clefs) ne conduit pas à la déduction d'autres clefs par l'intrus. Ces différentes propriétés seront détaillées au Chapitre 2.

Authentification La propriété d'authentification [67] d'un participant permet à une entité d'authentifier une autre entité, i.e. d'être assurée de l'identité de son correspondant. Ceci est généralement assuré par l'envoi d'un nonce encrypté par la clef publique de son correspondant. Celui ci, en recevant le message chiffré, le décrypte et renvoie le nonce. Ceci prouve qu'il a bien la clef privée qui lui est propre et qui lui a permis de décrypter le message et donc de prouver son identité.

Non répudiation Le principe de la propriété de non répudiation [67] est de fournir aux différentes parties des preuves que certaines étapes du protocole se sont produites. Par exemple, la non répudiation d'une transmission doit fournir au récepteur une preuve que le message a bien été envoyé par l'expéditeur réclamé. En outre, cette preuve doit être convaincante pour un tiers et non seulement pour le récepteur. Ceci a pour but d'empêcher une entité de nier avoir fait des actions précédentes.

1.1.5 Intrus, attaques

Intrus

Les propriétés de sécurité d'un protocole doivent être assurées dans un environnement hostile modélisé par un intrus. On désigne par intrus une entité malhonnête que l'on place entre les participants. Son but est de modifier le déroulement normal du protocole pour essayer de contredire une des propriétés de sécurité. Nous distinguons deux types d'intrus [67] :

- Intrus passif.
  L'adversaire ou l'intrus se contente d'écouter, d'enregistrer les messages circulant sur le réseau pour les analyser afin de déduire un secret. Par exemple, dans les protocoles d'établissement de clefs, l'intrus essaye de déduire la clef de session établie. Ce genre d'intrus menace la confidentialité des données.
- Intrus actif.
  L'adversaire ou l'intrus a les capacités de l'intrus passif et peut aussi supprimer, ajouter, ou changer la transmission sur le canal. Cet intrus modifie ou injecte les messages passés dans le réseau. Ce genre d'intrus menace non seulement la confidentialité des données, mais aussi l'authentification et l'intégrité des données.

Attaques

Pour attaquer un protocole cryptographique, différentes approches sont possibles. Une première approche se focalise sur les algorithmes cryptographiques. Une seconde approche suppose l'hypothèse du **chiffrement parfait**, i.e. la méthode de chiffrement est inviolable. Cette dernière approche repose sur la détection de failles logiques. La plus célèbre est celle de Needham-Schroeder [75]. Même si les informations transmises sont cryptées par un chiffrement parfait, les échanges ne sont pas sûrs sur un canal de communication public. Le protocole de Needham-Schroeder implique deux participants Alice et Bob. Le but de ce protocole est d'assurer l'authentification entre Alice et Bob, i.e. si Bob finit l'exécution de ce protocole, il pense qu'il est en train de communiquer avec Alice, et réciproquement. Le protocole est donné par la séquence de messages suivante :

$$\begin{array}{lrcl} 1 & Alice \longrightarrow Bob & : & \{A, N_a\}^p_{K_b} \\ 2 & Bob \longrightarrow Alice & : & \{N_a, N_b\}^p_{K_a} \\ 3 & Alice \longrightarrow Bob & : & \{N_b\}^p_{K_b} \end{array}$$

À l'étape 1, Alice initialise la communication en envoyant son identité A et un nonce (nombre aléatoire) généré fraîchement N_a, tous les deux encryptés par la clef publique de *Bob*. À la deuxième étape, en recevant le message $\{A, N_a\}^p_{K_b}$, Bob utilise sa clef privée pour déchiffrer le message reçu pour obtenir le nonce N_a de Alice. Il génère lui aussi un nonce N_b et encrypte ces deux nonces par la clef publique de Alice en composant le message $\{N_a, N_b\}^p_{K_a}$. À la réception de ce message à la dernière étape, Alice le déchiffre en utilisant sa clef privée pour obtenir le nonce N_b de Bob. Elle envoie ce nonce à Bob en le cryptant par la clef publique de celui-ci. Quand Alice reçoit le message $\{N_a, N_b\}^p_{K_a}$ et comme les nonces sont connus uniquement par elle et Bob, elle déduit que Bob lui a répondu. D'une manière similaire, quand Bob reçoit le message $\{N_b\}^p_{K_b}$, il déduit que Alice lui a répondu. Ceci permet de les authentifier : quand Alice reçoit un message contenant N_a ou N_b, elle en déduit qu'il vient de Bob et inversement. Une quinzaine d'années après la publication de ce protocole, une attaque logique a été trouvée sur ce protocole par G. Lowe [61]. Il s'agit de la faille connue sous le nom de l'homme au milieu (*man-in-the-middle attack*). Cette attaque est donnée par la Figure 1.5 : Alice (A) commence spontanément une conversation avec l'intrus (I). L'intrus se sert de ce premier message pour se faire passer pour Alice auprès de Bob (B). Celui-ci répond donc à Alice. Alice, reconnaissant son nonce N_a, pense que I vient de lui répondre. Elle lui renvoie donc le nonce N_b, que l'intrus n'aurait pas dû connaître. L'intrus termine alors le protocole avec Bob qui croit avoir parlé à Alice. Une variante du protocole a vu le jour en incluant l'identité de B à la fin du deuxième message transmis, i.e. Bob envoie $\{N_a, N_b, B\}^p_{K_a}$ au lieu de $\{N_a, N_b\}^p_{K_a}$. Cependant, une **attaque de type** a été découverte pour cette variante, i.e. un message d'un type particulier (ici une identité) a été interprété en tant que message d'un autre type (ici un nonce). Une autre variante du protocole a alors été introduite en changeant la position de l'identité dans le message, i.e. Bob envoie le message $\{B, N_a, N_b\}^p_{K_a}$.

Comme autre attaque classique, citons l'**attaque par rejeu**. Elle consiste à rejouer des messages envoyés précédemment dans la même session du protocole ou dans d'autres sessions de ce même protocole. Un exemple intuitif de ce type d'attaques est de considérer un intrus qui enregistre des paquets et les réexpédie textuellement, à un serveur d'authentification. Si, par chance, il a intercepté des paquets contenant une séquence complète d'authentification, il peut éventuellement se retrouver connecté sous l'identité qu'il a volée. D'autres listes d'attaques, spécifiques aux protocoles destinés à établir une clef entre deux participants, peuvent être trouvées dans [67].

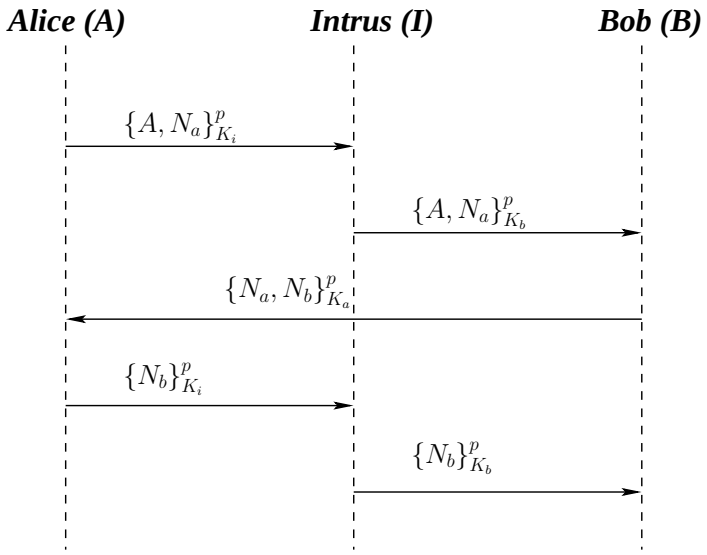


FIG. 1.5 – Attaque sur le protocole de Needham-Schroeder

1.2 Formalisation des protocoles cryptographiques

Il existe deux approches de modélisation de protocoles cryptographiques : le modèle symbolique et le modèle cryptographique. Le modèle cryptographique décrit les protocoles comme étant des envois de bits entre les différents participants du protocole, et cherche à assurer qu'aucune information secrète n'est divulguée. Ce modèle est plus proche de l'implantation du protocole. Le modèle symbolique consiste à abstraire les bits échangés entre les participants en les représentant par des termes, sur lesquels nous raisonnons pour vérifier le protocole. Dans ce manuscrit, nous nous intéressons à la vérification symbolique des protocoles cryptographiques. Le modèle symbolique a été décrit par Dolev et Yao dans [45] et a fait l'objet de plusieurs études. Son avantage principal est d'assurer l'automatisation de la vérification des protocoles. Nous rappelons que dans ce modèle, on suppose l'hypothèse de chiffrement parfait, i.e. il n'est pas possible de déduire un message à l'intérieur d'un message chiffré, sauf si on connaît la clef de déchiffrement.

Nous considérons aussi l'intrus de **Dolev Yao**, i.e. un intrus qui contrôle entièrement le canal de communication et peut agir sous respect de l'hypothèse de chiffrement parfait. Il peut donc modifier, supprimer, enregistrer les messages. Il peut aussi se faire passer pour un participant du protocole comme il peut initialiser des sessions de ce protocole.

Il existe plusieurs modèles pour spécifier un protocole cryptographique, selon les méthodes utilisées pour la vérification. Nous distinguons principalement deux modèles : celui de référence *Alice-Bob* et celui à base de rôles.

1.2.1 Modèle de référence : *Alice-Bob*

Comme introduit en Section 1.1.1, un protocole cryptographique est décrit intuitivement par des échanges de messages entre un ensemble d'entités. La syntaxe la plus simple et la plus intuitive est la notation *Alice-Bob*, couramment utilisée dans la littérature. Une liste de protocoles cryptographiques exprimés dans cette syntaxe figure dans [34]. Une syntaxe *Alice-Bob* décrit les actions effectuées par les entités dans le cas d'une exécution normale, i.e. sans

intervention de l'intrus. Une session du protocole est un échange de messages de la forme :

$$A \longrightarrow B : M$$

Cet échange de messages décrit l'envoi du message M par A à B où A et B sont deux agents. Les messages de base sont des noms d'agents (*Agents*), des atomes (*Atomes*) qui contiennent aussi les clefs symétriques atomiques, des nonces (*Nonces*), ou des clefs utilisées pour le chiffrement asymétriques (*ClefsAsym*). Un message M ($M \in MsgesAB$) est construit à partir de ces messages de base selon la grammaire suivante :

$$\begin{array}{rl} MsgesAB ::= & Agents \,|\, Atomes \,|\, Nonces \,|\, ClefsAsym \\ & \langle MsgesAB, MsgesAB \rangle \,|\, \{MsgesAB\}_{MsgesAB} \end{array}$$

où $\langle \_, \_ \rangle$ désigne la concaténation et $\{-\}_-$ désigne le chiffrement. Un protocole est un ensemble de connaissances initiales, i.e. des messages connus dès le début, pour chaque agent, et un ensemble de sessions différentes. Comme exemple de spécification dans cette syntaxe, nous pouvons citer la spécification des échanges de messages du protocole de Needham-Shroeder décrite en Section 1.1.5. La syntaxe du modèle *Alice-Bob* est simple et intuitive mais elle n'est pas expressive. En effet, elle ne décrit ni comment les participants vont procéder pour composer les messages qu'ils vont envoyer, ni comment les agents vont reconnaître les messages reçus, i.e. quelles parties de ces messages, devront-ils reconnaître.

1.2.2 Modèle de *rôles*

Un modèle plus précis, couramment utilisé, est le modèle à base de rôles. Un rôle est défini par un ensemble de *pas* représentant une action élémentaire du participant jouant ce rôle. Cette action associe à un message reçu la réponse correspondante envoyée par le participant en question. Un protocole est un ensemble de rôles. Nous nous focalisons maintenant sur les différences entre le modèle par rôles et celui *Alice-Bob*. Une différence majeure réside dans l'introduction des variables dans la construction de messages. En effet, les messages attendus par les participants peuvent contenir des variables pour représenter les connaissances acquises par le participant. L'ensemble des variables utilisées dans les différents pas de protocole représentent les connaissances acquises par les participants lors de l'exécution du protocole. Cet ensemble est noté Var. Cette notion permet de remédier à la difficulté rencontrée par le modèle *Alice-Bob*, i.e. celle d'identifier quelles connaissances possède un participant à un certain moment de l'exécution du protocole. En outre, avec le modèle de rôles et la notion de variables, il est possible d'explorer les attaques où l'intrus profite du fait qu'un participant n'a pas décomposé en totalité le message reçu. Les parties non examinées sont des variables que l'intrus peut remplacer par des messages qu'il construit pour aboutir à une attaque.

Exemple 1.2.2.1 *Nous revenons au protocole de Needham-Schroeder décrit en Section 1.1.5. Nous avons deux rôles pour ce protocole : un premier rôle joué par Alice et un deuxième par Bob. Nous spécifions ces deux rôles comme suit :*

Alice				**Bob**			
A_1	$Init$	$\Rightarrow$	$\{A, N_a\}^p_{K_b}$	B_1	$\{A, Y\}^p_{K_b}$	$\Rightarrow$	$\{Y, N_b\}^p_{K_a}$
A_2	$\{N_a, X\}^p_{K_a}$	$\Rightarrow$	$\{X\}^p_{K_b}$	B_2	$\{X\}^p_{K_b}$	$\Rightarrow$	End

Dans cette spécification, X et Y sont des variables, K_a et K_b sont des clefs, N_a et N_b sont des nonces et $Init$ et End sont des atomes. L'atome $Init$ permet d'initier la communication, i.e.

dans cet exemple, Alice commence l'exécution du protocole. L'atome End permet de terminer l'exécution du protocole. Ici, l'exécution termine lorsque Bob reçoit le dernier message. Nous nous focalisons maintenant sur le pas A_2. Alice attend dans ce pas, un message encrypté par sa clef publique, contenant son nonce N_a. Ce message contient aussi un autre nonce qu'elle ne connaît pas, d'où l'utilisation de la variable X. À la réception d'un tel message, Alice compose un message encrypté par la clef publique de Bob, contenant le nonce qu'elle vient de recevoir et qui est modélisé par X. Les différents pas de cette spécification sont partiellement ordonnés : A_1 doit précéder A_2 et B_1 précède B_2.

Une exécution du protocole correspond à un ordre total, compatible avec l'ordre partiel précédent, sur l'ensemble des pas de ce protocole, i.e. si un pas p_1 précède un autre pas p_2 selon l'ordre partiel alors cette précédence est maintenue selon l'ordre de l'exécution. Ensuite, toutes les variables sont instanciées par des messages dans cette exécution.

Le problème de l'insécurité, i.e. l'existence d'une attaque de secret peut être traduite par l'existence d'une exécution où :

- pour chacun des pas, l'intrus arrive à composer le message attendu à partir de ses connaissances acquises des messages envoyés pendant tous les pas précédents ;
- à partir des connaissances acquises de toute l'exécution (des messages envoyés de tous les pas), l'intrus peut composer le secret.

1.3 Méthodes de vérification des protocoles cryptographiques

Comme cité en introduction de la Section 1.2, nous nous intéressons dans ce manuscrit aux approches symboliques de vérification de protocoles cryptographiques. Les protocoles cryptographiques sont difficiles à vérifier car ils présentent plusieurs sources d'indécidabilité. En effet, ils peuvent impliquer des nombres non bornés de sessions, de participants, ou de clefs et de nonces générés au cours d'une session. Ils peuvent aussi manipuler des messages de taille variable non bornée. En outre, dans une exécution de protocole, l'intrus peut générer un nombre arbitraire de nonces ou de clefs. Même avec quelques restrictions, le problème d'insécurité, i.e. déterminer si une donnée bien particulière reste secrète ou non pour le protocole considéré, reste indécidable. Par exemple, Durgin et al. ont montré dans [47] que le problème d'insécurité est indécidable même en bornant la taille des messages. En outre, même en bornant le nombre de nonces générés dans une exécution du protocole (même sans nonces), le problème d'insécurité reste encore indécidable [38]. Diverses méthodes de vérifications ont donné naissance à plusieurs outils dédiés à la vérification automatique des protocoles cryptographiques. Dans cette section, nous allons parcourir en bref deux principales approches adoptées pour analyser un protocole cryptographique. Plus de résultats de décidabilité/indécidabilité peuvent être trouvés dans [39].

1.3.1 Falsification des protocoles cryptographiques

Il existe des méthodes ayant comme objectif la falsification des protocoles cryptographiques, i.e. la recherche d'éventuelles attaques sur ces protocoles. Cette approche est motivée par le fait que la plupart des attaques implique un nombre réduit de sessions et de messages.

En bornant le nombre de sessions, de participants et la taille des messages, l'analyse devient finie. Le nombre d'états accessibles à partir de l'état initial est alors borné. Le protocole est modélisé par une machine à nombre d'états fini et les propriétés par des formules de logique temporelle. Il est donc possible d'utiliser des outils de model-checking dans ce cas, tels que FDR [46], Murphi [70], ou Brutus [35]. Ces outils ont permis la découverte d'attaques intéressantes. Par

exemple, C'est grâce à FDR, que la fameuse attaque sur le protocole Needham Schroeder a été trouvée.

D'autres approches ont restreint le nombre de sessions mais la taille des messages reste non bornée. Dans ce contexte, la méthode utilisée repose sur les techniques symboliques, i.e. représenter les états concrets par des états symboliques. Une première méthode a été proposée par Huima [52] à base de réécriture. Le problème de secret est prouvé NP-complet dans [89]. Dans le même contexte, le problème de secret est formalisé comme un problème de résolution de contraintes [69] : la propriété de secret est violée si l'intrus arrive à résoudre une séquence de contraintes. Les contraintes sont tout d'abord mises en forme normale ou résolues, en utilisant un ensemble de règles de simplification. Les contraintes finales peuvent alors être testées pour la satisfaisabilité d'une manière plus simple que pour les contraintes initiales.

Comme exemples d'outils implémentant ce genre de méthodes, nous pouvons citer AtSe [102] qui fait partie de l'outil AVISPA [4] dédié à la vérification des protocoles cryptographiques, comportant quatre outils.

1.3.2 Validation des protocoles cryptographiques

Les méthodes précédentes sont efficaces pour trouver des attaques. Cependant, si elles n'ont pas trouvé d'attaques, rien ne garantit la correction du protocole, i.e. l'absence d'attaques sur ce protocole. En effet, si aucune attaque n'est trouvée en fixant le nombre de sessions, il peut exister une attaque pour ce même protocole en considérant un nombre de sessions plus grand. Ainsi, quelques travaux ont abordé la vérification des protocoles cryptographiques à nombre non borné de sessions de différentes manières. Parmi les principales approches utilisées, nous trouvons les procédures de semi-décision ou celles utilisant des approximations ou des abstractions, ou encore celles considérant des classes particulières de protocoles.

Méthodes de semi-décision Ces méthodes sont utilisables si le modèle de protocoles n'est pas restreint et la vérification est automatique. En contrepartie, ces méthodes peuvent ne pas donner de résultat : soit elles ne terminent pas, soit elles nécessitent une intervention pour prouver quelques lemmes.

Parmi les premiers outils dans cette catégorie, nous trouvons NRL [64]. Cet outil implémente une méthode de preuve pour les protocoles cryptographiques sans aucune restriction, en explorant symboliquement les différents états du protocole. Cependant, l'utilisateur doit intervenir pour assurer la terminaison de cette exploration en prouvant à la main certains lemmes.

Il existe d'autres techniques plus autonomes comme celle de Casrul [16] ou de Athena [92]. Les deux méthodes utilisent des explorations symboliques des états du protocole. Le premier (Casrul) les effectue en avant : de l'état initial au but. Quant au deuxième (Athena), il les fait en arrière. Si les protocoles ne sont pas sûrs alors ces méthodes terminent toujours. Sinon, elles peuvent boucler indéfiniment. Dans la même catégorie, nous avons Scyther [41]. Cet outil peut soit trouver des attaques soit prouver la correction du protocole pour un nombre non borné de sessions. Il utilise un algorithme de raffinement de patterns, permettant de représenter des ensembles infinis de traces. La terminaison est garantie dans la plupart des cas. Par exemple, en pratique, pour la librairie SPORE [1], un résultat général de falsification/vérification a été obtenu pour 80% de ces protocoles et pour le reste (20%), Scyther fournit juste une vérification bornée.

Méthodes par approximations Une deuxième approche pour prouver la correction des protocoles cryptographiques sans fixer le nombre de sessions, est d'utiliser des approximations

ou des abstractions. Ces méthodes utilisent toujours des explorations symboliques des états du protocole, mais en effectuant des approximations sur des ensembles d'états considérés, afin d'assurer la terminaison. Ces approximations doivent être correctes : si le protocole est sûr avec ces approximations, alors il est sûr sans elles. Cependant, ces approximations peuvent introduire de fausses attaques. Un premier exemple de ces approximations a été introduit par Monniaux, et Genet et Klay respectivement dans [72, 49] en utilisant les automates d'arbres pour faire des approximations sur les connaissances de l'intrus. La méthode termine modulo quelques heuristiques permettant de prouver la sécurité de certains protocoles. Comme outil implémentant cette méthode, citons Timbuk [50].

Dans la même catégorie, nous trouvons l'outil Hermes [21] qui considère des approximations des messages protégés et des messages potentiellement connus par l'intrus. La preuve de secret se résume alors en un test d'incompatibilité de ces deux ensembles. Cet outil a permis de prouver le secret de plusieurs protocoles de [34].

D'autres méthodes utilisent les clauses de Horn pour modéliser les règles des protocoles. Ces méthodes permettent d'abstraire les sessions et l'ordre d'exécution des règles des protocoles. Un premier travail se basant sur cette approche est celui de Weidenbach [103]. Comme outil implémentant cette approche, citons ProVerif [12].

Toutes ces méthodes terminent souvent mais peuvent détecter de fausses attaques.

Classes de décidabilité Certains travaux se sont focalisés sur des classes bien particulières des protocoles. Le premier résultat de décidabilité était celui de [45] où les protocoles ping-pong étaient considérés, i.e. des protocoles impliquant des participants qui ne peuvent qu'appliquer des opérateurs unaires sur le dernier message reçu et envoyer le résultat. Or, cette restriction est trop sévère. D'autres travaux ont considéré des classes beaucoup plus riches. Certaines méthodes ont considéré des modèles avec un nombre non borné de nonces et ont permis d'obtenir des résultats de décidabilité. soit par considération d'un système fortement typé (e.g. [62]), soit par considération d'un système de marquage (*tagging*) empêchant l'unification de différents sous-messages encryptés (e.g. [87]). D'autres se sont intéressés à d'autres classes mais considérant un nombre borné de nonces. Par exemple, Comon-Lundh et Cortier ont montré dans [37] que le problème est décidable pour un modèle de protocoles sans nonces tel qu'au plus une partie inconnue du message est testée et/ou copiée à chaque transition. Le modèle utilisé dans ces travaux est celui des clauses de Horn. Dans le même contexte de clauses de Horn, d'autres ont utilisé la marquage pour assurer la terminaison pour l'outil ProVerif [14].

Plus de détails sur ces résultats de décidabilité ainsi que d'autres travaux peuvent être trouvés dans [39].

1.4 Contenu de la thèse

Dans cette thèse, nous nous intéressons à l'étude des protocoles de sécurité paramétrés et plus particulièrement aux protocoles paramétrés par le nombre de participants, i.e. aux protocoles de groupe. Notre but est de proposer des méthodes pouvant être automatisées afin d'assurer la vérification automatique de ces protocoles. Traiter les protocoles de groupe c'est aborder essentiellement deux points. D'abord, traiter des propriétés de sécurité spécifiques et ensuite, un nombre arbitraire de participants.

Pour présenter notre travail, nous organisons ce manuscrit en trois parties. La première partie consiste en un état de l'art sur les protocoles de groupes et leur vérification. Cette partie regroupe aussi nos expérimentations préliminaires, vérifiant quelques protocoles de groupe.

Dans la deuxième partie, nous présentons notre première contribution qui consiste à traiter la problématique de la modélisation et la vérification des propriétés de sécurité de la classe des protocoles considérés. Dans la troisième partie, nous traitons la problématique du nombre non borné de participants des protocoles de groupe et la vérification des protocoles cryptographiques avec listes paramétrées. Cette partie comprend un nouveau résultat de décidabilité pour une classe de protocoles de groupe.

État de l'art

Cette partie, composée des deux chapitres 2 et 3, a pour but de présenter et de synthétiser les travaux de recherche dans le domaine de la vérification de protocoles paramétrés.

Protocoles paramétrés, Chapitre 2

Le but de ce chapitre est de définir les protocoles paramétrés et plus spécifiquement les protocoles paramétrés par le nombre de participants impliqués. Nous rappelons quelques définitions de ces protocoles dans la littérature. Pour sécuriser les communications entre les différents participants des protocoles de groupe, ces participants ont besoin d'une clef commune. Cette clef est d'abord établie puis mise à jour suivant l'évolution du groupe. D'où la nécessité d'une gestion de clefs. Nous détaillons dans ce chapitre, les différentes approches pour cette gestion. Ensuite, les protocoles en général sont conçus pour satisfaire certaines propriétés de sécurité. Nous survolons donc, ces propriétés en les classant selon leurs dépendances par rapport à l'évolution de la structure du groupe .

Vérification de protocoles de groupe, Chapitre 3

Ce chapitre a pour but de survoler les travaux effectués pour la vérification des protocoles de groupe. Nous commençons par déduire les problèmes que peuvent rencontrer la vérification de tels protocoles. Nous présentons ensuite les travaux théoriques. Dans cette catégorie, quelques résultats de décidabilité [60, 99, 59, 57] pour certaines classes de protocoles de groupe ont vu le jour. Nous survolons ensuite quelques travaux sur la recherche d'attaques et qui ont mené à la découverte d'attaques intéressantes. Finalement, nous présentons les travaux que nous avons effectué dans le but de retrouver des attaques sur quelques protocoles et d'en découvrir de nouvelles. Ces travaux montrent comment spécifier en HLPSL [25] certains protocoles de groupe afin de les vérifier par l'outil AVISPA [4].

Modélisation des propriétés de protocoles de groupe

Cette partie, composée du Chapitre 4, a pour but de présenter notre première contribution dédiée à la recherche d'attaques sur les protocoles de groupe en tenant compte des différentes propriétés de sécurité.

Détection de types d'attaques sur les protocoles de groupe, Chapitre 4

Nous commençons dans cette partie par présenter le modèle que nous avons proposé pour les protocoles de groupe et plus généralement pour les protocoles dits contribuants. Ce modèle est appelé modèle de services et est basé sur le travail de Pereira et Quisquater [81]. Ce modèle permet de décrire un protocole de groupe, d'étudier ses caractéristiques et ses propriétés de sécurité, et donc d'identifier les différents types d'attaques possibles. Le modèle est présenté sous forme d'un système à plusieurs composantes qui intéragissent entre elles. Nous décrivons ces différentes

composantes ainsi que les caractéristiques de notre modèle. Nous présentons après la formalisation des propriétés de sécurité des protocoles de groupe. Cette étude nous a permis de détecter différents types d'attaques. Nous avons appliqué le modèle sur plusieurs protocoles, tels que A-GDH.2 [8], SA-GDH.2 [8], Asokan-Ginzboorg [6] et Bresson-Chevassaut-Essiari-Pointcheval [22], ce qui nous a permis de mettre en évidence différents types d'attaques possibles sur chacun. Enfin, nous synthétisons les résultats trouvés par cette approche. Ce travail présenté a été publié dans [30] (élu meilleur article) et est paru dans la revue REE [31].

Nous présentons ensuite, une stratégie de recherche d'attaques pour les protocoles d'accord de clefs et plus généralement pour les protocoles dits contribuants. Cette stratégie combine le modèle de service et la résolution de contraintes. Par application de cette stratégie sur des protocoles tels que Asokan-Ginzboorg, GDH.2 et A-GDH.2, nous avons pu retrouver des attaques, trouver de nouvelles attaques et même trouver des attaques générales (pour les protocoles de GDH.2 et A-GDH.2) impliquant n'importe quel nombre de participants.

Nous commençons par décrire notre procédure de recherche d'attaques en définissant l'entrée de notre méthode ainsi que ses différentes étapes et fonctionalités. Nous détaillons ensuite, la gestion des contraintes, utilisée par notre méthode en introduisant la notion d'arbre de contraintes permettant d'assurer la construction d'une éventuelle attaque pour un protocole étudié. Enfin, nous synthétisons les résultats de l'application de notre méthode pour certains protocoles. Ce travail a fait l'objet de l'article [32] et du chapitre de livre [33].

Vérification de protocoles avec nombre non borné de participants ou avec listes paramétrées

Cette partie, composée des deux chapitres 5 et 6, a pour but de présenter notre deuxième contribution dédiée à la décidabilité du problème d'insécurité d'une classe de protocoles de groupes.

Un modèle pour la vérification de protocoles paramétrés, Chapitre 5

Le but de ce chapitre est de proposer un modèle synchrone pour les protocoles paramétrés avec un nombre fini de sessions. Ce modèle est une extension de modèles classiques de vérification de protocoles tels que [89] afin de pouvoir manipuler des listes de messages dont la longueur est donnée comme paramètre. Mis à part les protocoles de groupe dont le paramètre est le nombre de participants, les listes paramétrées apparaissent aussi dans les protocoles de web services où les messages sont semi-structurés.

Nous commençons ce chapitre par la présentation du modèle synchrone, ainsi que la transformation permettant le passage d'un protocole d'origine à un autre synchrone afin de pouvoir le traiter dans notre modèle. Ce modèle se base essentiellement sur l'ajout d'un opérateur spécial noté *mpair* pour pouvoir gérer les listes. Nous montrons que l'ajout naïf de cet opérateur mène à l'indécidabilité du problème d'insécurité. D'où l'introduction de la classe de protocoles bien-tagués avec clefs autonomes. Nous introduisons aussi, les prédicats auxiliaires et leurs sémantiques. Ces prédicats expriment la construction de messages à partir des connaissances de l'intrus. Ils sont utilisés pour former le système de contraintes dont la satisfaisabilité est équivalente à l'existence d'attaques pour le protocole considéré. Nous proposons ensuite un ensemble de règles d'inférence correctes et complètes pour vérifier la sécurité pour la classe de protocoles bien tagués avec clefs autonomes et ceci en présence d'un intrus **actif**.

Décidabilité pour les protocoles paramétrés bien tagués à clefs autonomes, Chapitre 6

Ce chapitre propose une procédure de décision pour la classe introduite dans le chapitre précédent, i.e. la classe des protocoles bien tagués avec clefs autonomes. Nous énonçons les résultats obtenus pour cette procédure et plus particulièrement le résultat de décidabilité obtenu pour la classe considérée.

La procédure de vérification considérée se base essentiellement sur l'application des règles introduites dans le chapitre précédent, et modélise le problème d'insécurité des protocoles. Nous prouvons que l'ensemble de nos règles est correct et complet. Nous justifions que notre modèle est effectivement une extension des modèles classiques des protocoles cryptographiques en prouvant que, dans le cas d'absence d'indices ou de *mpair* pouvant générer des indices, nos règles d'inférence terminent. Nous montrons aussi la terminaison pour la classe des protocoles bien tagués avec clefs autonomes. Nous prouvons ensuite que la normalisation d'un système de contraintes par notre ensemble de règles conduit à des contraintes en forme normale dont on peut tester la satisfaisabilité. Nous obtenons ainsi un résultat de décidabilité pour la classe considérée, celle des protocoles bien tagués avec clefs autonomes. Nous donnons à la fin de ce chapitre des exemples testés par application de notre procédure. Notons que, en considérant des protocoles paramétrés dont le nombre de participants n n'est pas fixé, si notre procédure finit par ne pas trouver d'attaques, alors le protocole considéré n'admet aucune attaque pour toute valeur du paramètre n. Notons aussi que notre procédure permet aussi de traiter les protocoles avec **clefs composées**.

Les travaux des chapitres 5 et 6 ont fait l'objet des publications dans [28] et [29] (détaillé dans le rapport technique [27]).

2

Protocoles de groupe

Sommaire

2.1 Introduction

Le but de ce chapitre est de définir les protocoles paramétrés et plus spécifiquement les protocoles paramétrés par le nombre de participants, appelés protocoles de groupe. Un survol de ces définitions est donné en Section 2.2. Pour sécuriser les communications entre les différents participants des protocoles de groupe, ces participants ont besoin d'une clef commune. Cette clef est d'abord établie puis mise à jour suivant l'évolution de la structure du groupe. D'où la nécessité d'une gestion de clefs. Cette notion est détaillée en Section 2.3. Les protocoles sont en général conçus pour satisfaire certaines propriétés de sécurité. Nous survolons en Section 2.4 ces propriétés, en les classant selon leurs dépendances par rapport à l'évolution de la structure du groupe. Finalement, nous donnons en bref une autre caractéristique intéressante liée aux protocoles de groupe et qui les distingue de ceux à deux participants. Il s'agit de l'impact des agents malhonnêtes sur le fonctionnement de ces protocoles. Notons aussi que la plupart de ces notions seront illustrées dans ce chapitre par différentes applications réelles.

2.2 Protocoles paramétrés

L'objectif de cette section est de survoler dans un premier temps en Section 2.2.1 les définitions existantes d'un système paramétré afin de pouvoir définir un protocole paramétré.

Nous définissons par la suite en Section 2.2.2 la classe des protocoles paramétrés, considérés dans ce manuscrit, ainsi que leurs caractéristiques.

2.2.1 Classe de protocoles paramétrés

Nous commençons tout d'abord par voir ce que signifie un système paramétré dans la littérature. Nous définissons après la notion de protocoles paramétrés avec quelques exemples d'utilisation de ces protocoles.

Système paramétré Nous pouvons définir un système paramétré comme une collection d'un nombre arbitraire de composants agissant l'un sur l'autre par l'intermédiaire d'une liaison synchrone ou asynchrone. Dans [106], un système paramétré consiste en une composition en parallèle de n processus symétriques, exécutants tous, le même programme. Dans le même esprit, d'après [84], un système paramétré consiste en un nombre arbitraire de processus à états finis. Ces systèmes paramétrés peuvent utiliser des structures de données telles que les tableaux qui ont aussi une longueur paramétrée selon le nombre de processus.

Considérons maintenant un réseau où un certain nombre de processus concurrents interagissent entre eux. Généralement, ce nombre de processus concurrents n'est pas connu en avance dans une telle situation. L'intérêt d'une telle caractéristique est de concevoir des protocoles qui peuvent fonctionner correctement pour n'importe quel nombre de processus concurrents. Ces protocoles sont alors nommés protocoles paramétrés par considération du nombre de processus.

Un exemple classique de système paramétré est de considérer les algorithmes distribués où n utilisateurs partagent une ressource et exécutent un protocole pour assurer l'accès mutuel exclusif à cette ressource. La propriété d'exclusion mutuelle à une ressource, pour un système donné de n processus, exprime le fait que, tout i et j de $\{1, .., n\}$, tant que le processus i exécute une partie de sa section critique et donc utilise la ressource partagée, le processus j où $j \neq i$ n'exécute aucune partie de sa section critique et donc n'utilise pas la ressource partagée. Dans ce cadre de protocoles, nous pouvons citer le protocole paramétré de Peterson [83]. Ce protocole assure l'utilisation mutuelle exclusive d'une ressource donnée qui est partagée entre n processus, où n est le paramètre du protocole paramétré et peut être non borné.

En résumé, un système paramétré représente une famille de systèmes à états finis utilisant des type définis récursivement tels que les chaînes ou les arbres. Le paramètre de ses systèmes est généralement le nombre de processus dans ce système qui peut être arbitrairement grand. En effet, les systèmes paramétrés sont des systèmes pouvant avoir un nombre **arbitraire** de processus **identiques** mis en parallèle, chaque processus étant un système fini. Ces systèmes sont omniprésents dans plusieurs contextes. Par exemple, les protocoles gérants les réseaux doivent fonctionner quelque soit le nombre de composantes de ces réseaux.

Protocole paramétré Dans le même esprit, dans le cadre des protocoles cryptographiques définis en Section [**?**], un protocole paramétré peut être défini comme étant un protocole mettant en cause un nombre arbitraire de participants. Néanmoins, au contraire des systèmes paramétrés où les processus sont identiques, les participants du protocole paramétré peuvent ne pas effectuer

des actions similaires. En d'autres termes, ces participants peuvent avoir des comportements différents vis à vis d'un même message reçu. Ainsi, nous pouvons les classer dans un premier temps en deux catégories :

- les protocoles symétriques.
  Dans ces protocoles, chaque participant exécute des actions similaires. En effet, en recevant un certain message, un participant traite ce message et envoie un autre message comme réponse. Dans un protocole symétrique, tous les participants ont le même comportement vis à vis d'un même message reçu, i.e. ils envoient la même réponse, modulo les informations privées utilisées lors du traitement du message reçu. Par exemple, pour la première phase du protocole GDH [8], en recevant une liste de m messages, chacun des participants exponentie chacun de ces messages par sa contribution personnelle, construit et envoie la liste composée de $m-1$ messages modifiés, du message m et du message m modifié.
- les protocoles asymétriques.
  Dans ce cas, certains participants ne se comportent pas comme les autres participants vis à vis d'un même message reçu. Un exemple de ce genre de situation se présente dans des protocoles où il y a deux sortes de participants : une entité centrale et le reste des participants. Cette entité centrale peut être définie dès le départ sans tenir compte de la topologie de l'ensemble des participants. Il s'agit généralement d'un leader ou d'un serveur qui a pour rôle de contrôler l'échange de messages entre ces participants ou bien la résolution des requêtes de ces participants. Cette entité centrale peut aussi être définie par rapport à la topologie de l'ensemble des participants. Dans ce cas, elle est généralement le dernier élément de la chaîne représentant cet ensemble de participants. Comme exemple, nous pouvons citer les protocoles Tanaka Sato [97] ou Asokan-Ginzboorg [6]. Le premier protocole met en cause un serveur et un ensemble de participants. Le serveur reçoit des requêtes des autres participants, et les traite selon le type de ces requêtes. Dans le deuxième protocole, le dernier participant joue le rôle d'un leader qui collecte les contributions des autres participants afin de calculer une clef du groupe.

Une autre caractéristique qui peut différencier les protocoles paramétrés des autres protocoles est l'impact du paramètre qui est le nombre de participants sur la définition et la structure elles-mêmes du protocole. Nous considérons donc dans ce cas deux catégories de protocoles paramétrés :

- la structure du protocole dépend du paramètre (*Dependant Structure protocols*). Dans ce cas, la définition même du protocole dépend de la valeur de n qui est le nombre de participants impliqués dans le protocole. En d'autres termes, la définition du protocole diffère d'une valeur de n à une autre. Par exemple, le nombre de phases constituant le protocole diffère pour différentes valeurs de n. Comme exemple de cette catégorie nous pouvons citer le protocole de signature de contrat de Garay-MacKenzie (GM) [48] et plus précisément le sous-protocole principal de ce protocole (*Main*). Ce sous-protocole consiste en un échange de promesses entre les signataires (participants du protocole). Il est composé de n niveaux qui sont définis récursivement. En outre, chaque niveau de récursion utilise un niveau de promesses différent et plus fort que l'ancien. Il est donc évident que la structure de sous-protocole diffère pour différentes valeurs du paramètre n.
- la structure du protocole ne dépend pas du paramètre (*Independant Structure protocol*). Le nombre de phases du protocole ou bien le déroulement du protocole ne fait donc pas intervenir le paramètre. Ils sont toujours valables pour n'importe quelle valeur du paramètre. Par exemple, le protocole Asokan-Ginzboorg [6] admet exactement quatre phases indépendamment du nombre de participants impliqués. Dans la première phase, le leader envoie à l'ensemble des participants une clef d'encryption. La deuxième phase consiste

à collecter les contributions de ces différents participants. En troisième phase, le leader envoie la clef de groupe concaténée à d'autres informations relatives à chacun des participants. Finalement, la quatrième phase est la confirmation du bon calcul de la clef du groupe par les différents participants. L'exécution de ce protocole pour n'importe quelle valeur du paramètre du protocole ne change que le nombre de messages échangés dans chacune des phases mais ne change en aucun cas la structure de ce protocole.

Autre paramétrage. Une autre sorte de paramétrage des protocoles est de considérer des structures de données complexes récursives telles que les tables ou les listes dont la longueur est paramétrée, mais indépendamment du nombre de participants impliqués. Ces protocoles sont présents dans plusieurs applications. Nous présentons dans ce qui suit quelques exemples.

Comme premier contexte, nous pouvons citer le cas d'un participant qui reçoit une liste dont il ne connaît pas la longueur. Il va traiter les messages composant la liste et va construire un message à base des informations qu'il vient d'acquérir. Il va par la suite envoyer le message qu'il a composé. Un exemple pratique de cette situation est de considérer un client qui a besoin d'informations concernant un service particulier. Il contacte un intermédiaire pour récupérer les informations nécessaires. Ce dernier lui envoie alors la liste des offres concernant ce service. Le client compose alors sa requête finale et la transmet à l'intermédiaire. Ce cas suppose implicitement l'existence de plusieurs serveurs fournissant chacun le service voulu du client. Le paramètre dépend donc implicitement du nombre de participants du protocole. Cependant, ce cas peut aussi être indépendant des participants. En effet, ce client peut demander directement à un certain serveur plusieurs services fournis par ce seul serveur.

Ce genre de situation peut aussi se présenter dans le cas d'une édition collaborative d'un même document. Des modifications sont faites en même temps sur plusieurs parties de ce même document. Le document peut être alors modélisé comme une liste de plusieurs parties en cours de modification par plusieurs participants.

Dans le même contexte de fichiers, nous pouvons aussi citer le cas d'un serveur envoyant à un client un fichier de taille arbitraire. Ce fichier est sous forme d'une liste de morceaux de fichiers.

Nous revenons maintenant aux protocoles paramétrés par le nombre de participants. Dans ce type de protocoles, n agents participent à l'exécution du protocole, n étant un paramètre. Sauf information explicite, nous considérons tout au long de ce manuscrit ce type de protocoles.

2.2.2 Protocoles de groupe

Nous rappelons que les protocoles de groupe sont des protocoles impliquant un nombre arbitraire de participants. Généralement ce nombre n'est pas connu à l'avance. Dans un tel protocole, un groupe de participants échange des informations afin d'aboutir à un but commun. Comme exemple de buts, nous pouvons citer la déduction d'une clef commune [7] entre ces participants leur permettant d'échanger de futures communications d'une manière sécurisée, ou encore la signature d'un certain contrat entre eux [48]. Nous pouvons aussi citer les protocoles d'échange équitable multi-parties [9] qui sont une généralisation de l'échange équitable à deux parties. Dans ces protocoles à deux participants A et B, A possède un message m_A et B possède un message m_B avec l'intention pour A d'obtenir m_B et pour B d'obtenir m_A. Le protocole a pour but de garantir l'équité : soit A obtient m_B et B obtient m_A, soit aucun d'eux n'obtient le message attendu.

Les protocoles de groupe sont présents dans la plupart des applications. Grâce au développement

des technologies suite à l'évolution de l'Internet au cours de la dernière décennie, plusieurs applications ont vu le jour. Comme exemples de telles applications, nous pouvons citer les audio ou vidéo conférences, l'enseignement à distance, la télévision à la demande ou encore les jeux de groupe interactifs.

L'aspect de groupe existe aussi dans le domaine de la défense (militaires, pompiers...). Les protocoles de groupe sont alors nécessaires pour assurer les communications entre les forces militaires ou la coordination des communications entre les forces civiles dans le cas des intempéries ou des missions. La sécurisation de ces communications devient dans ces cas un objectif primordial.

Toutes ces applications ont un point commun. Il s'agit toujours d'un groupe de membres dont il faut sécuriser les communications même dans un environnement hostile. Cependant, la structure de ce groupe n'est pas toujours fixe vu qu'il peut subir des mouvements, soit provenant de ses membres, soit provenant de l'extérieur. En effet, une entreprise peut changer, selon ses besoins, son personnel en recrutant de nouveaux employés ou en licenciant d'autres. De même, un groupe de soldats partant en mission peut perdre des membres ou encore être renforcé par de nouveaux volontaires. D'une manière générale, les opérations que nous pouvons avoir dans un groupe peuvent mettre en cause une seule entité ou encore un ensemble d'entités affectant la stabilité du groupe.

Opérations pour une seule entité Considérons le contexte d'un groupe existant de participants et suivons son évolution dans le temps.

Un groupe peut être altéré par la présence d'un nouveau prétendant voulant adhérer à ce groupe. Généralement, cette opération s'effectue entre le nouveau prétendant et le responsable du groupe. Ce groupe peut aussi subir une exclusion d'un membre. La sortie de ce membre du groupe peut être soit par consentement entre les deux parties impliquées ou de la part du chef du groupe seulement.

Opérations pour un ensemble d'entités Nous nous intéressons maintenant aux opérations susceptibles d'être effectuées par un ensemble d'entités. Un résumé de ces opérations est illustré par la Figure 2.1.

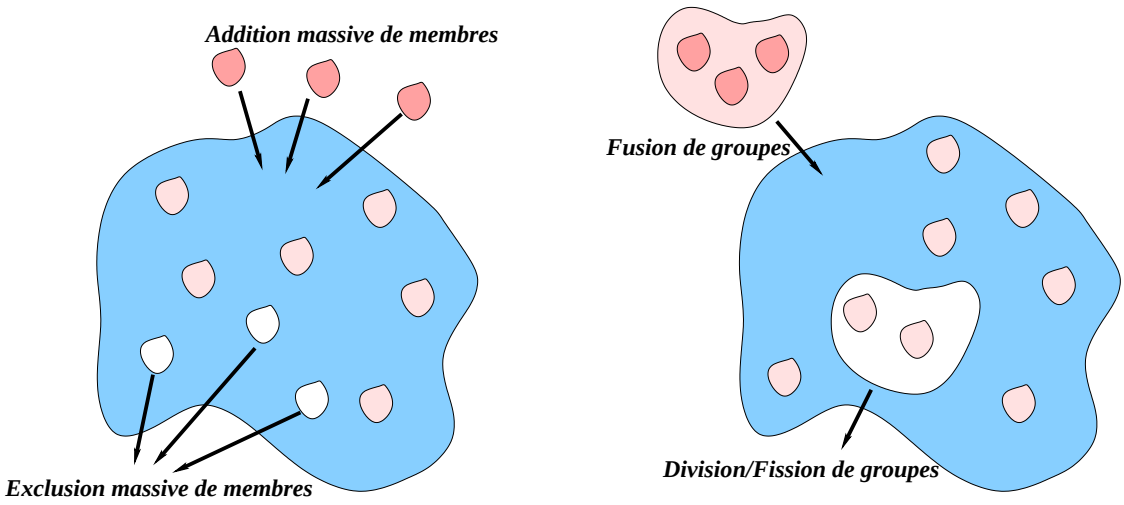


FIG. 2.1 – Opérations relatives à un ensemble d'entités

Considérons un groupe existant. Il se peut que plusieurs nouveaux membres veulent joindre à la fois ce groupe. Ces membres n'ont aucun lien entre eux, c'est-à-dire ils n'appartenaient pas auparavant à un même groupe ; on assiste alors à une addition massive de membres (*Mass Join*). De la même manière, un groupe peut exclure plusieurs membres en même temps. On assiste alors à une exclusion massive de membres.

Dans le même esprit, deux groupes peuvent fusionner. Dans ce cas, pour deux groupes A et B, leur fusion permet d'étendre les communications des membres de A vers B et inversement. De la même manière mais cette fois dans le sens inverse, un sous-groupe peut quitter le groupe. On parle alors de fission de groupe (*Group Fission*). Une autre situation peut apparaître aussi lorsqu'un groupe monolithique se divise en petits sous-groupes. On parle dans ce cas de division de groupe (*Group Division*).

L'évolution d'un groupe peut se résumer entre des ajouts ou des retraits de membres de ce groupe. Quelle que soit l'opération qui altère le groupe, la communication de ce groupe doit continuer d'être sécurisée. En particulier,

- les anciens membres d'un groupe ne doivent pas avoir accès aux communications du nouveau groupe ;
- les nouveaux membres du groupe ne doivent pas accéder aux communications passées.

Les opérations mentionnées ci-dessus n'affectent que la composition du groupe. Néanmoins, le groupe peut aussi changer de structure, i.e. les membres peuvent changer de position au sein du groupe. Ce cas se présente lorsque le groupe est représenté sous forme de classes hiérarchiques, e.g. dans les domaines militaires ou de forces civiles. Les communications se font des niveaux supérieurs vers les niveaux inférieurs. De nouvelles conditions de sécurité sont alors introduites. Par exemple, les membres d'une classe peuvent accéder aux communications mettant en cause les membres de cette classe. Ils peuvent aussi accéder aux communications de toutes les classes inférieures à leur classe. Cependant, ils ne peuvent jamais accéder aux communications de classes strictement supérieures. Il est évident alors que le changement de structuration de groupe entraîne des changements considérables de procédures de sécurisation des communications du groupe. Comme exemple d'opérations de changement de structure, nous pouvons citer la montée de classe, la descente de classe. Dans la montée de classe, un membre se déplace d'une classe à une autre classe supérieure. Il a donc accès aux communications de la classe cible ainsi que ses classes inférieures. De la même manière mais dans le sens inverse, on peut aussi assister à une descente de classe. Dans ce cas, un membre se déplace vers une classe cible strictement inférieure à sa classe d'origine.

En résumé, un groupe doit tout d'abord sécuriser sa communication. Cette sécurisation se fait généralement par l'établissement d'une clef commune entre les membres de ce groupe, appelée clef du groupe. Ensuite, vu que le groupe peut changer de structure ou de composition, la communication doit être une fois de plus sécurisée, et donc la clef doit être modifiée. D'où le besoin de la gestion de clefs.

2.3 Gestion de clefs

Dans une communication de groupe, quand on veut envoyer un message à tous les membres du groupe, il est plus efficace d'utiliser une communication multicast, i.e. utiliser un seul canal

de communication pour tout le groupe, que d'utiliser un canal (unicast) pour chacun de ces membres. Cependant, une telle communication doit affronter plusieurs types de menaces quelque soit le mode de communication utilisé (synchrone ou asynchrone). Nous pouvons citer comme exemple pour le mode asynchrone, la télévision à la demande. Pour le mode synchrone, nous pouvons penser à la vidéo-conférence, utilisée dans le cadre des meetings d'une entreprise ou d'enseignement à distance.

Dans un environnement hostile, les membres de ces groupes ont besoin de sécuriser leurs communications contre toute personne extérieure. Par exemple, dans le cas de la société offrant des programmes de télévision à la demande, vu que ce service est payant, tout individu n'ayant pas payé pour ce service ne doit pas être capable d'avoir accès à ces programmes. De la même manière, pour le cas d'un meeting international, l'entreprise doit être sure de ne divulguer aucun résultat de son meeting à une entreprise concurrente.

Un moyen de sécuriser cette communication est d'utiliser les algorithmes de chiffrement, connus pour leurs bienfaits pour une communication à deux participants. Les messages de cette communication sont alors protégés par encryption en utilisant une clef choisie, connue dans le contexte des protocoles de groupe sous le nom de clef du groupe. Ainsi, seuls les participants connaissant la clef du groupe (et donc les membres légitimes du groupe) peuvent accéder au message original.

Le besoin d'utilisation de cette clef du groupe implique sa génération et sa distribution à toute personne ayant le droit d'accès à cette clef. Généralement, les groupes ont recours à l'utilisation d'une clef par session. Une clef de session est définie en [67] comme une clef dont l'utilisation est restreinte pour un temps court comme le temps d'une connexion (ou la session), après lequel toute trace de cette clef est éliminée. Selon [67], l'intérêt d'utiliser ces clefs de session est tout d'abord de limiter l'ensemble des textes chiffrés disponibles qui seraient utilisées pour d'éventuelles attaques. Ensuite, la clef de session permet d'éviter le stockage à long-terme d'un large nombre de clefs secrètes distinctes, en créant des clefs uniquement à la demande. En outre, cette notion de clef de session permet de créer une indépendance de communication entre les différentes sessions et applications. La génération de cette clef est effectuée par le biais des protocoles d'établissement de clefs. Ces protocoles résultent généralement des secrets partagés qui servent à dériver les clefs de sessions. Une fois que cette clef de groupe est établie, elle doit suivre l'évolution du groupe. On parle alors, plus généralement, de la gestion de clef du groupe, i.e. son établissement et sa mise à jour. Les protocoles de gestion de clefs du groupe sont notés GKMP pour *Group Key Management Protocols*. Nous nous basons dans cette section sur les études faites dans [63, 85].

Il existe plusieurs approches pour la gestion de clefs de groupe. Nous les résumons en trois classes principales : centralisées, décentralisées et distribuées. Ces approches ainsi que quelques exemples de protocoles les illustrant sont donnés en Figure 2.2.

2.3.1 Approche centralisée

Dans cette approche, nous considérons les protocoles de distribution de clefs de groupe. Un protocole de distribution de clefs est défini dans [63], comme étant un protocole d'établissement de clef où une partie crée ou obtient une valeur secrète et la transfère secrètement aux autres. Cette définition est basé sur les définitions du Chapitre 12 de [67],

D'après cette définition, une seule entité choisit la clef du groupe et la transfère secrètement aux autres membres du groupe. Cette entité est appelée contrôleur du groupe, notée GC pour *Group Controller*. Elle peut être un membre du groupe comme elle peut ne pas faire partie du groupe. Par exemple, un TTP (*Trusted Third Party*) peut être l'entité centralisée qui choisit la

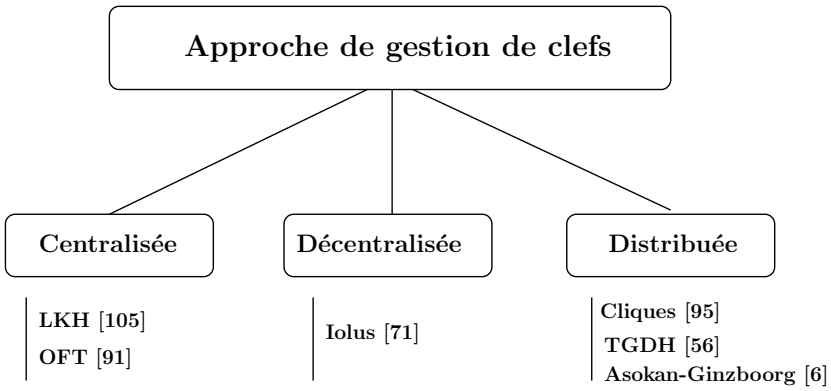


FIG. 2.2 – Les approches de gestion de clefs

clef d'un groupe au nom de tous les membres de ce groupe.

Cependant, avoir une seule entité pour contrôler tout le groupe peut avoir des conséquences graves. En effet, cette approche ne permet pas le passage à l'échelle vu que, si le groupe est grand, l'entité centrale aura du mal à le contrôler. En outre, cette entité centrale présente une cible privilégiée pour les attaques : une fois que cette entité a un problème, tout le groupe est affecté. Le groupe devient alors vulnérable puisque les clefs ne sont plus générées et distribuées.

Le défi pour l'approche centralisée est d'assurer la protection de la clef durant sa phase de transfert. Ceci suppose l'existence de canaux de communication protégés entre cette entité et chacun des membres du groupe. Parmi les mécanismes utilisés pour arriver à cette fin, i.e. sécuriser la phase de distribution de la clef du groupe, un mécanisme d'*hiérarchie de clefs* peut être utilisé. Comme exemple de protocoles utilisant ce mécanisme, nous citons LKH pour *Logical Key Hierarchy* ([105]). Dans ce protocole, le contrôleur de groupe maintient un arbre de clefs. Les noeuds de cet arbre présentent des clefs d'encryption de clefs (KEK). Les feuilles de cet arbre correspondent aux membres du groupe et chaque feuille présente la KEK du membre correspondant. Chaque membre du groupe détient la KEK correspondant à sa feuille et les KEKs correspondant à tous les noeuds appartenant au chemin entre son noeud parent et la racine de l'arbre. La clef du groupe est celle qui se trouve à la racine de l'arbre. Le mécanisme des hiérarchies de clefs est essentiellement utilisé pour gérer le dynamisme du groupe vu que la modification de la structure d'un groupe revient à la modification de la structure de l'arbre logique des clefs ; plusieurs protocoles ont été proposé pour ce problème (LKH : [105], OFT : [91]). Par exemple, pour le cas du protocole LKH, l'ajout d'un membre est géré par la génération de nouvelles clefs logiques du chemin allant du noeud à ajouter à la racine de l'arbre.

En effet, la Figure 2.3 montre l'impact de l'ajout d'un membre M_4 à un groupe, géré par le protocole LKH. Suite à cet événement, toutes les clefs allant du nœud de M_4 à la racine doivent être changées. D'où, la transmission d'un message de renouvellement des KEKs contenant toutes les clefs à modifier encryptées par les KEKs de leurs noeuds fils. Quand M_4 rejoint le groupe, il reçoit une clef secrète k_4. Les nouvelles KEKs (k'_{34}, k'_{14} et k') sont générées et encryptées par les KEKs de leurs noeuds fils correspondants. Par exemple, k'_{34} est encryptée par k_3 et k_4.

2.3.2 Approche décentralisée

Dans cette approche, la gestion du groupe est assurée par plusieurs contrôleurs de sous-groupes (connus sous le nom de clusters). L'intérêt d'avoir un contrôleur pour chaque cluster

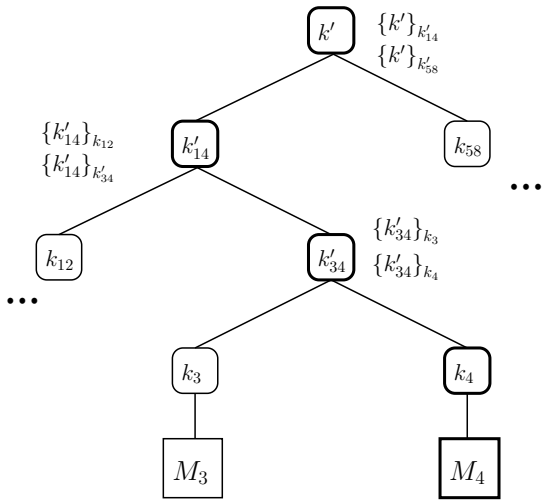


FIG. 2.3 – Ajout d'un membre M_4 par le protocole LKH

est de minimiser l'effet d'une modification de la structure du groupe (e.g. ajout ou retrait d'un membre) sur la clef du groupe. En effet, l'ajout d'un membre au groupe clusterisé n'affecte pas la totalité du groupe mais plutôt le cluster auquel s'ajoute ce membre.

Comme exemple de protocoles suivant cette approche, nous citons le protocole **Iolus** [71] (Voir Figure 2.4). En effet, dans ce protocole, le groupe est divisé en petits sous-groupes (ou

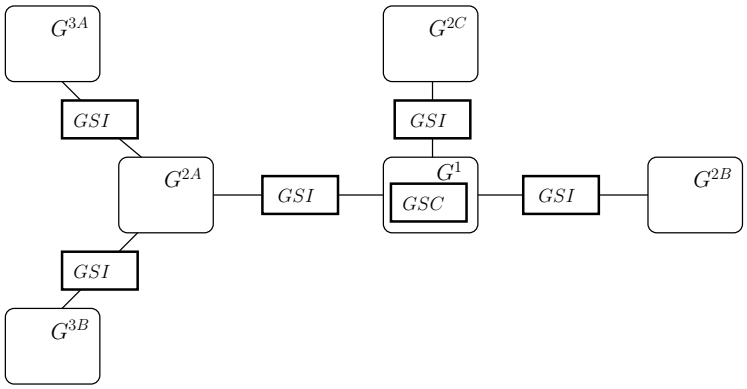


FIG. 2.4 – Clusterisation pour le protocole Iolus

clusters). Chaque cluster est géré par un agent de sécurité de groupe (GSA). Les GSAs sont groupés en un seul niveau supérieur contrôlé par le contrôleur de sécurité du groupe (GSC). Les GSIs sont les intermédiaires de sécurité de groupe qui contrôlent chaque cluster. Il est à noter que les GSIs et les GSCs sont tous les deux des GSAs. Le protocole Iolus utilise des clefs indépendantes pour chacun des clusters. Ainsi, la dynamique du groupe n'affecte qu'un seul sous-groupe du groupe. En outre, si un GSA n'est plus fonctionnel, seul son sous-groupe est affecté, ce qui contribue à la tolérance aux fautes de tout le système. Néanmoins, le GSA a plus de fonctions qu'un contrôleur central d'une approche centralisée. En effet, mis à part son travail de contrôle, il doit assurer la transition entre les clusters vu que les clefs d'encryption de données diffèrent d'un cluster à un autre. Ceci peut entraîner le dysfonctionnement des GSAs vu qu'ils

forment de vrais goulots d'étranglement.

2.3.3 Approche distribuée

Dans cette approche, nous considérons les protocoles d'accord de clefs de groupe. Un protocole d'accord de clefs est défini dans [63], en se basant sur les définitions du chapitre 12 de [67], comme étant un protocole d'établissement de clef où un secret partagé est dérivé par deux ou plusieurs parties en fonction d'informations contribuées par ou associées à chacun d'eux. Idéalement, chacun de ses participants ne doit pas être capable de prévoir la valeur résultante de ce secret partagé.

D'après cette définition, un protocole d'accord de clefs est un protocole permettant à un groupe de participants de générer une clef dont la valeur est basée sur leurs contributions et sans avoir à établir auparavant une clef partagée. La génération de cette clef s'effectue par un simple échange de messages sur un réseau pouvant être sous le contrôle total d'un intrus. Ces échanges de messages renferment les contributions liées à la clef de groupe qui ne peut pas être déterminée à l'avance. En outre, la clef obtenue à la fin de l'exécution de ce genre de protocole ne peut être connue que par les membres du groupe. Les protocoles d'accord de clefs pour plus de deux entités sont connus sous le nom de protocoles d'accord de clefs de groupe ou GKAP pour *Group Key Agreement Protocol*.

Comme exemple d'approche distribuée nous pouvons citer [20] où une clef de conférence est à générer à partir des contributions de tous les membres du groupe. Cette clef de groupe a pour valeur $f(C_1, .., C_n)$ pour une fonction f et pour une contribution du participant i (C_i). Mis à part le leader du groupe, les membres du groupe envoient en clair leurs contributions à tout le groupe. Le leader du groupe encrypte sa contribution avec la clef publique de chacun des participants et envoie la liste de ces messages composés à tout le groupe. Tous les membres du groupe peuvent alors déduire la clef du groupe.

Dans le même style, nous avons aussi le protocole de Asokan-Ginzboorg [6], où un groupe de participants tente d'établir une clef de session afin de sécuriser leur prochaine communication, et ce, en connaissant un mot de passe connu par tous ces membres au départ.

Plusieurs protocoles d'accord de clef de groupe peuvent être vus comme des extensions de protocoles d'accord de clefs à deux participants. Par exemple, la suite de protocoles **Cliques** [8, 7, 95] est une extension du protocole de Diffie-Hellman (DH) [44] afin de pouvoir supporter les opérations de groupe (voir Section 2.2.2). Il s'agit d'un ensemble de protocoles fournissant un accord de clefs entièrement authentifié et contribuant pour les groupes dynamiques. Parmi ces protocoles, il y a des protocoles de génération de clefs, appelés IKA (*Initial Key Agreement*), et les protocoles auxiliaires, appelés AKA (*Auxilliary Key Agreement*), permettant de suivre le dynamisme du groupe. Comme exemple de protocoles de IKA, nous trouvons GDH (Group DH), A-GDH (Authenticated GDH) ou SA-GDH. Pour les protocoles de AKA, nous citons par exemple, le protocole GDH-MA utilisé dans le cas d'un ajout d'un membre.

La Figure 2.5 illustre l'exécution de deux protocoles de Cliques : le protocole GDH à trois participants, et le protocole de GDH-MA pour le cas d'un ajout d'un quatrième membre à ce groupe.

Dans la première partie de cette figure, modélisant l'exécution du protocole GDH à trois participants, les nonces r_1, r_2 et r_3 désignent respectivement les contributions des membres M_1, M_2 et M_3. La clef de groupe dans cette exécution est $\alpha^{r_1 r_2 r_3}$. Cette clef est générée respectivement

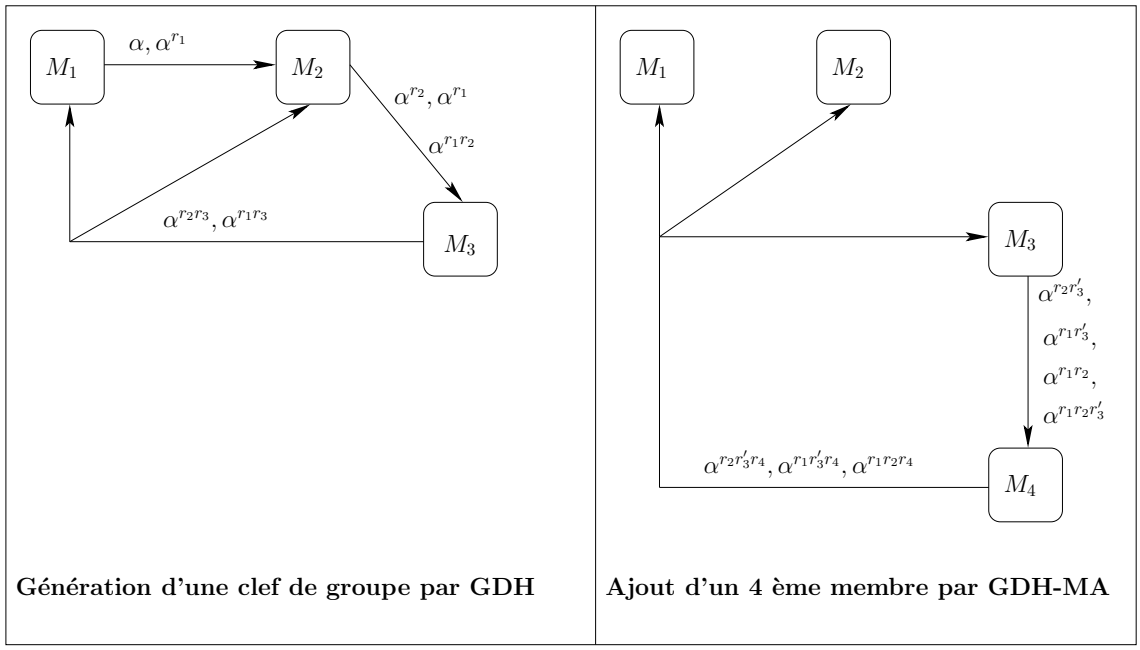


FIG. 2.5 – Génération de clefs et ajout d'un membre dans Cliques

par les membres M_1, M_2 et M_3 à partir des messages respectifs $\alpha^{r_2 r_3}$, $\alpha^{r_1 r_3}$ et $\alpha^{r_1 r_2}$ en les exponentiant par leurs contributions respectives (r_1, r_2, r_3). La deuxième partie de la Figure 2.5 représente l'ajout d'un quatrième membre M_4 au groupe établi par la première partie de cette même figure. Le participant M_3 utilise le dernier message qu'il a reçu durant la génération de la clef. Il exponentie les composants de ce message par une nouvelle contribution r'_3 et envoie le message au nouveau membre M_4. Ce dernier ajoute aussi sa contribution et envoie le résultat au reste du groupe qui va être capable de dériver la clef du groupe dont la valeur est $\alpha^{r_1 r_2 r'_3 r_4}$.

Il existe d'autres approches qui se basent sur le même protocole DH. Nous pouvons citer le protocole **Octopus** [11] qui est similaire à Cliques sauf qu'il utilise DH pour dériver une clef intermédiaire dans chacun des quatre sous-groupes formant le groupe principal. La clef du groupe sera dérivée après avoir échangé les clefs intermédiaires entre les différents leaders des sous groupes, où le leader désigne le responsable de la collecte des contributions.

D'autres approches ont combiné l'adoption de DH avec quelques techniques similaires à celles utilisées dans l'approche centralisée. Par exemple, le protocole **T-GDH** [56] utilise la même technique de hiérarchie de clefs que celle du protocole OFT (One-way Function Tree) de l'approche centralisée. Les membres du groupe utilisent donc un arbre binaire hiérarchique de clefs. Cependant, ces membres génèrent les clefs du niveau supérieur en utilisant DH au lieu d'utiliser des fonctions à sens unique comme le cas d'OFT.

2.3.4 Discussion

Chacune des approches de gestion de clefs a ses avantages et ses inconvénients. Pour l'approche centralisée, une seule entité centrale contrôle le groupe. C'est au contrôleur de choisir la clef du groupe et de la distribuer au reste du groupe. Avec une seule entité qui contrôle tout le groupe, cette entité présente un point attractif pour d'éventuelles attaques. Il suffit qu'elle fonctionne mal pour que tout le système s'écroule. Cette approche se confronte aussi au problème

de passage à l'échelle vu que, si le groupe devient trop grand, l'entité centrale aura du mal à le gérer.

Avec l'approche décentralisée, le groupe est divisé en petits sous-groupes contrôlés indépendamment par des GSAs. Cette distribution de contrôles entre les GSAs permet d'éviter l'échec central de l'approche centralisée puisque, pour affecter tout le groupe, il faut que tous les contrôleurs GSAs fonctionnent mal. Cependant, cette approche lève une question de confiance puisqu'on doit faire confiance à plusieurs contrôleurs au lieu d'un seul.

L'approche distribuée se caractérise par l'absence de contrôleur de groupe. Tous les membres contribuent à la clef de groupe qu'ils vont partager. La différence principale entre l'approche distribuée et celle centralisée est que, au contraire de la deuxième, dans la première, aucun membre du groupe n'est autorisé à choisir la clef du groupe au nom du reste du groupe. En outre, dans l'approche distribuée, il n'est pas nécessaire d'avoir des canaux sécurisés entre les participants, vu qu'il n'y a pas besoin de transferts sécurisés, comme pour la distribution de clefs. Cependant, pareil à l'approche centralisée, le passage à l'échelle n'est pas garanti pour l'approche distribué. En effet, la clef du groupe repose sur les contributions de tous les membres du groupe, ce qui demande beaucoup de calculs lorsque le nombre de contributions devient important.

2.4 Propriétés de sécurité

Toutes les approches de gestion de clefs décrites en Section 2.3 ont pour objectif de dériver une clef de groupe commune à tous les membres de ce groupe afin de sécuriser leurs communications. L'objectif de la section actuelle est de faire un survol sur les propriétés de sécurité liées à cette clef de groupe. La gestion de clef englobe tout d'abord l'établissement de cette clef et aussi sa mise à jour en tenant compte de l'évolution de la structure du groupe. En effet, lors d'un ajout d'un membre à un groupe ou la sortie d'un autre, la clef de groupe doit être changée. Les propriétés de sécurité que doit satisfaire un protocole de gestion de clefs doivent tenir compte de ces deux fonctionnalités, i.e. l'établissement de la clef et sa mise à jour. Nous introduisons dans cette section les propriétés de sécurité considérées dans les protocoles de groupe et plus spécifiquement pour les protocoles de gestion de clef de groupe, en tenant compte de ces deux fonctionnalités. En Section 2.4.1, nous identifions des propriétés inspirées de celles introduites pour les protocoles d'établissement de clefs à deux participants, mais qui restent aussi valables pour les protocoles de groupe. Nous introduisons après en Section 2.4.2, les propriétés de sécurité relatives à l'établissement de la clef du groupe pour un état stable du groupe, sans tenir compte de l'évolution de la structure de ce groupe. La plupart de ces propriétés étaient utilisées dans le cadre des protocoles de Cliques décrits en Section 2.3.3. Ensuite, nous décrivons en Section 2.4.3 quelques propriétés nécessaires pour le suivi de la dynamique d'un groupe. Finalement, nous donnons en Section 2.4.4 un résumé des propriétés que doit satisfaire un protocole de gestion de clefs.

2.4.1 Propriétés classiques

Certaines propriétés de sécurité introduites pour les protocoles cryptographiques en général (voir Section 1.1.4) sont aussi valables pour les protocoles de groupe et plus spécifiquement pour les protocoles d'accord de clefs de groupe. En outre, d'autres propriétés considérées pour les protocoles d'établissement de clefs à deux participants sont aussi valides pour les protocoles d'accord de clefs de groupe.

Nous nous focalisons dans un premier temps sur la phase de génération de la clef de groupe. Tout d'abord, la clef de groupe ne doit être connue que par les membres de ce groupe. Cette propriété est définie dans [55] dans le cadre des **protocoles d'accord de clefs**.

Définition 2.4.1.1 *Secret de la clef du groupe.*
Cette propriété garantit qu'il est impossible d'un point de vue calculatoire pour un intrus passif de déduire n'importe quelle clef du groupe.

Il est à noter que cette propriété ne concerne pas uniquement les protocoles d'accord de clefs mais plutôt tout protocole ayant pour objectif de générer ou de distribuer une clef pour sécuriser une communication entre plusieurs membres d'un groupe. Par exemple, dans l'accord de Diffie-Hellman (DH) et les autres protocoles qui se basent sur DH tel que GDH, la garantie de cette propriété se base essentiellement sur la difficulté des calculs (exponentiation...) effectués par un intrus passif pour découvrir la clef du groupe. Pour l'approche centralisée (les protocoles de distribution de clefs), la phase la plus délicate est celle de l'envoi de la clef du groupe par le contrôleur du groupe au reste du groupe. Généralement, la clef de groupe est déduite par les différents membres du groupe suite à des décryptions successives, ce qui assure naturellement le secret de leur clef commune.

Comme deuxième propriété, nous citons celle d'authentification implicite qui permet à une entité, dans le cadre des protocoles à deux participants, d'authentifier une autre entité, i.e. être assurée de l'identité de l'autre entité impliquée dans le protocole. D'après [67], cette propriété peut être décrite par la définition suivante :

Définition 2.4.1.2 *Authentification implicite.*
Cette propriété permet de garantir à une première partie que l'accès à la clef secrète ne soit possible ***que*** *pour une seconde partie bien déterminée (avec possibilité d'autres parties additionnelles de confiance).*

Dans le cadre des protocoles d'établissement de clefs de groupe, cette définition permet à un membre légitime d'un groupe de s'assurer qu'aucune autre personne, mis à part les autres membres, ne connaisse la clef du groupe. Cependant, cette propriété ne garantit pas que chaque membre du groupe finit par avoir la même clef du groupe. Notons que cette propriété est appelée implicite car elle ne dépend ni de la possession actuelle de la clef par la seconde partie ni de la connaissance de cette possession actuelle par la première partie.
Cette dernière caractéristique, i.e. la garantie de la possession de la clef de groupe par la seconde partie est assurée par la propriété de la confirmation de clef.

Définition 2.4.1.3 *Confirmation de la clef.*
Cette propriété permet de garantir à une première partie qu'une seconde partie possède actuellement une clef de session particulière.

Dans le contexte de protocoles de gestion de clefs de groupe, la propriété de confirmation de clefs permet à chaque membre du groupe d'être assuré que chacun des autres membres est actuellement en possession de la clef du groupe.

La conjonction de cette propriété avec celle d'authentification implicite mène à une autre propriété qui est l'authentification explicite définie dans [67] comme suit :

Définition 2.4.1.4 *Authentification explicite.*

Cette propriété est assurée quand les deux propriétés d'authentification implicite et de confirmation de la clef sont vérifiées.

La propriété d'authentification explicite permet de garantir que chaque participant identifié du groupe est connu pour avoir la clef actuelle du groupe.

Pour toutes les propriétés de sécurité définies ci-dessus, l'intrus tente de deviner ou de dériver la clef du groupe ou bien il tente de se faire passer pour un membre du groupe afin d'aboutir à cette même fin, i.e. récupérer la clef du groupe. Dans le contexte du groupe, l'intrus peut aussi avoir comme objectif de diviser le groupe en le poussant à dériver des clefs différentes les uns des autres. Ceci n'est possible que dans le cas des protocoles d'accord de clefs. Ce genre d'attaques est connu sous le non d'attaques de contrôle de clefs, où l'intrus essaye d'influencer la valeur résultante de la clef calculée de groupe. Cette attaque est définie ainsi :

Définition 2.4.1.5 *Contrôle de la clef du groupe.*
Au contraire des protocoles de transport de clefs où une partie choisit une valeur de la clef et la transfère secrètement à une autre partie, dans les protocoles d'accord de clefs, la clef est dérivée des informations communes et aucune partie du groupe ne peut contrôler ou prévoir la valeur de cette clef.

Il est à noter que les attaques de contrôle de clefs sont surtout présentes en présence des participants malhonnêtes, i.e. des participants qui souhaitent nuire au bon fonctionnement du protocole et donc au bon calcul de la clef du groupe.

Nous revenons maintenant aux objectifs de la gestion de clefs de groupe et nous rappelons que, mis à part l'établissement de la clef du groupe, cette gestion doit aussi tenir compte de l'évolution de la structure du groupe, et donc modifier cette clef selon les besoins. En particulier, pour l'analyse de protocoles d'établissement de clefs [67], l'impact du compromis d'autres clefs (des différents types) doit être considéré même si un tel compromis n'est pas normalement prévu. En effet, il faut considérer l'effet du :

– compromis des clefs secrètes à long terme,
– compromis des clefs de sessions antérieures.

Dans un premier temps, nous considérons l'impact du compromis des clefs secrètes à long terme. D'où l'introduction de la propriété de secret futur.

Définition 2.4.1.6 *Secret futur parfait (PFS).*
Un protocole satisfait la propriété de secret futur parfait, connue aussi sous le nom de break-backward protection*, si le compromis des clefs à long terme n'entraîne pas le compromis des clefs de sessions passées.*

La propriété de secret futur parfait sert à verrouiller d'une manière sécurisée le trafic du passé. En effet, un intrus qui vient d'acquérir une clef à long terme ne doit pas être capable de revenir sur des clefs de sessions qui sont déjà passées et de les compromettre. Il ne peut pas donc avoir accès aux différentes communications passées. Néanmoins, d'après cette propriété, le compromis des clefs à long terme ne peut pas empêcher un intrus actif de les utiliser pour l'espionnage des sessions futures. La propriété de secret futur parfait peut être garantie généralement si la clef de session ne dépend pas des clefs à long terme. Par exemple, cette propriété peut être garantie en générant des clefs de sessions selon l'accord de Diffie-Hellman puisque, dans ce cas, les exponentielles sont basées sur des clefs à court-terme, i.e relatives à la session en question.

Dans certains environnements, la probabilité de compromettre une clef de session est beaucoup plus grande que celle de compromettre une clef à long terme. Il est donc nécessaire de

protéger les sessions futures contre le compromis des clefs de sessions qui sont beaucoup moins protégées que les clefs à long terme. Nous considérons donc la définition suivante d'attaques qui se basent sur le compromis des clefs des sessions passées :

Définition 2.4.1.7 *Attaques à clefs connues.*
Un protocole est vulnérable à des attaques à clefs connues si le compromis des clefs de sessions passées permet soit à un intrus passif de compromettre des clefs de sessions futures, soit à un intrus actif d'espionner de sessions futures.

Sans parler d'intrus, nous donnons ici quelques exemples de l'intérêt de cette propriété. Considérons, dans un premier temps, l'exemple d'une société qui vend ses programmes sur Internet. Si un membre vient de s'abonner à partir d'une date bien déterminée alors, bien que ce membre fasse partie du groupe à partir de la date de son abonnement, il ne peut pas accéder aux différents programmes fournis par la société avant cette date d'abonnement. De la même manière, supposons qu'un participant P a participé au meeting organisé par une entreprise faisant des meetings entre ses différentes branches à distance. Ce participant P ne doit pas être capable d'avoir accès aux communications d'éventuels prochains meetings. En effet, il n'y a aucune garantie que ce participant aura le droit d'assister aux prochains meetings, vu qu'il y aurait possibilité de changement de statut et donc de changement de composition de cette entreprise. Ainsi, même si la clef de session d'une session courante est connue, cela ne doit en aucun cas permettre de dériver une clef de session future, et donc d'avoir un accès aux communications futures.

2.4.2 Propriétés d'un groupe statique

Dans le cadre de la suite de protocoles de Cliques (voir Section 2.3.3), quelques propriétés de sécurité ont été définies dans [8] puis reprises dans [79]. Nous rappelons que le cadre est les protocoles d'*accord de clef* de groupe, où certain nombre de participants sont censés générer une clef commune entre eux à base des contributions de chacun d'eux. Nous retrouvons la même notion d'authentification implicite que celle de la Définition 2.4.1.2 pour les protocoles à deux participants, i.e. l'assurance qu'aucun membre externe au groupe n'acquiert la vue de la clef du groupe d'un participant. La vue de la clef de groupe d'un participant désigne la valeur de la clef de groupe qu'il vient de calculer. Cette propriété est décrite par la définition suivante :

Définition 2.4.2.1 *Authentification implicite de la clef du groupe.*
Un protocole vérifie la propriété d'authentification implicite (IKA) si, après avoir terminé son rôle dans une session du protocole, chaque membre est assuré qu'aucun autre membre extérieur au groupe ne peut avoir sa vue de la clef de groupe (i.e, la valeur de la clef qu'il a calculée).

Cependant, cette propriété ne signifie, ni que tous les membres de groupe ont une connaissance de la clef de groupe à la fin du protocole, nor qu'ils se mettent d'accord sur sa valeur.

La confirmation de la clef dans un protocole d'établissement de clefs à deux parties fournit une évidence que la clef est détenue par une partie bien déterminée. Cette propriété a été définie dans le cadre des protocoles à deux participants dans la Définition 2.4.1.3. Dans un groupe, chaque participant doit être assuré que tous les autres membres du groupe possèdent la même clef partagée.

Définition 2.4.2.2 *Confirmation de la clef du groupe.*
Un protocole satisfait la confirmation de la clef du groupe si chaque partie est assurée que son groupe possède actuellement cette clef.

Cette propriété suppose qu'une clef secrète partagée fait partie des connaissances de tous les autres membres du groupe.

Dans le cadre de protocoles de distribution de clefs, la propriété d'intégrité de la clef a été introduite en [100]. Elle exprime le fait que la clef distribuée ne peut pas être modifiée par l'intrus, et donc, les informations composant cette clef proviennent seulement des participants légitimes du protocole. Dans [7], Ateniese et al. étendent cette définition pour exprimer le fait que la clef du groupe établie est déduite à partir des contributions de **tous** les membres du groupe.

Définition 2.4.2.3 *Intégrité.*

Un protocole contribuant satisfait la propriété d'intégrité si chaque partie est assurée que sa clef secrète n'est fonction que des contributions de tous les membres du groupe. En particulier, les contributions des éléments externes au groupe ne sont pas tolérées même si elles ne fournissent aucune information utile pour un intrus.

Un moyen de garantir cette propriété est, par exemple, d'imposer à chaque participant de signer et de transférer aux autres une attestation (*statement*) certifiant sa participation au protocole. Notons que cette propriété peut être garantie si la propriété d'authentification explicite est satisfaite.

Ateniese et al. introduisent dans [7] la notion d'accord de clef contribuable. Il s'agit des protocoles d'accord de clefs, où chaque participant contribue de manière équitable à l'établissement de la clef du groupe et garantit sa fraîcheur. Ils introduisent par la suite la notion d'accord de clef contribuable vérifiable :

Définition 2.4.2.4 *Contribution vérifiable.*

Chaque partie du protocole est assurée de la contribution de chacune des autres parties du groupe concernant la clef à partager. Cette propriété vérifie l'intégrité, mais l'intégrité seule, ne peut pas donner la contribution vérifiable.

Notons que cette propriété suppose que personne ne peut prédire la clef du groupe à calculer. Les protocoles d'accord de clefs doivent assurer aussi cette propriété même en présence de participants malhonnêtes.

Ateniese et al. définissent en plus l'authentification complète de la clef de groupe. Il s'agit d'une propriété des protocoles d'accord de clefs où tous les participants calculent la même clef seulement si chacun de ces participants a contribué à son calcul. Cette propriété peut être vue comme étant la combinaison des deux propriétés : la contribution vérifiable (Définition 2.4.2.4) et l'authentification explicite (Définition 2.4.1.4). Elle implique seulement que l'intrus ne peut pas altérer la structure du groupe en excluant certains membres de la participation à la clef du groupe.

2.4.3 Propriétés d'un groupe dynamique

Pereira [79] reprend la propriété de secret futur parfait (PFS) en la détaillant plus. Il définit deux variantes de cette propriété selon que l'on a manipulé ou modifié aucun message des sessions passées (Définition 2.4.3.1), ou bien que l'on a manipulé quelques messages (Définition 2.4.3.2). Cette différenciation de cas n'était pas nécessaire dans le cas de PFS puisque cette propriété a été définie au départ dans le contexte des protocoles à deux participants. Dans un tel contexte, l'intrus n'avait pas la possibilité de manipuler des messages puisque les deux participants ne

seraient pas d'accord sur une clef de session unique. Il n'y aura pas donc d'ultérieures communications permettant à l'intrus de manipuler d'éventuels messages. Dans le cadre des protocoles de groupe, i.e. à plusieurs participants, la manipulation d'un message peut altérer la vue de la clef d'un participant particulier alors que les autres participants du groupe calculent la même clef. Cependant, cette manipulation n'empêche pas ces derniers participants de communiquer entre eux avec cette clef commune, et donc de fournir des messages que l'intrus peut manipuler.

Ainsi, la première variante, correspondant au cas de manipulations d'aucun message du passé comme suit :

Définition 2.4.3.1 *Secret futur complet.*
Un protocole satisfait la propriété de secret futur complet si le compromis des clefs à long terme n'entraîne pas le compromis des clefs de sessions passées en supposant qu'aucun message de sessions antérieures du protocole n'a pas pu être manipulé.

Quand à la deuxième variante, elle correspond à la manipulation de certains messages dans le passé, et est définie par :

Définition 2.4.3.2 *Secret futur individuel.*
Un protocole satisfait la propriété de secret futur individuel si le compromis des clefs à long terme n'entraîne pas le compromis des clefs de sessions passées, et ce en supposant que quelques membres du groupe ont fait l'objet d'attaques dans le passé tout en gardant les autres membres du groupe non affectés.

Dans un scénario de la Définition 2.4.3.2, le membre attaqué est contraint d'être exclu du groupe en générant une clef de session différente de toutes les autres. Le reste du groupe a une même vue de la clef de groupe et donc peut communiquer entre eux. Le participant attaqué se trouve passif : il ne fait que recevoir les messages échangés entre les autres sans même pouvoir les déchiffrer ou participer à cette communication.

Le travail [79] détaille aussi la notion d'attaques à clefs connues en se basant sur l'idée que les clefs de session ne sont pas uniquement les secrets de sessions. En outre, pour les protocoles d'accord de clefs, ces clefs de session sont basées essentiellement sur les contributions des membres du groupe. Ces contributions des différents membres sont alors elles aussi des secrets relatifs aux sessions.

Définition 2.4.3.3 *Attaques à secrets de sessions connus.*
Un protocole est vulnérable aux attaques à secrets de sessions connus si le compromis des secrets de sessions passées permet, soit à un intrus passif de compromettre des clefs de sessions futures, soit à un intrus actif d'espionner des sessions futures.

Remarquons ici que, d'une manière triviale, si un protocole est vulnérable à des attaques à clefs de sessions connues alors ce protocole est vulnérable à des attaques à secrets de sessions connus.

D'autres variantes de propriétés ont été définies dans [55]. Elles sont aussi valables pour les *protocoles d'accord de clefs*. Elles dépendent de la dynamique du groupe et se focalisent sur l'impact des clefs d'une certaine session sur les clefs des autres sessions. Nous commençons par les définitions des secret passé et futur faibles. Ces deux propriétés ont comme caractéristiques de supposer que l'intrus n'est autre qu'un membre du groupe. La première se focalise sur les droits que peut avoir un nouveau membre du groupe.

Définition 2.4.3.4 *Secret de passé faible (BSW).*
Cette propriété garantit que les clefs passées du groupe ne doivent pas être découvertes par les nouveaux membres du groupe.

La deuxième propriété, i.e. le secret futur faible, se focalise sur un ancien membre du groupe.

Définition 2.4.3.5 *Secret de futur faible (FSW).*
Cette propriété garantit que les nouvelles clefs restent hors de portée des anciens membres du groupe.

L'intérêt des deux propriétés (secrets futur et passé faibles) est de verrouiller les droits des membres du groupe. Un éventuel membre futur d'un groupe ne doit pas avoir accès aux communications passées de ce groupe et donc n'a pas le droit de connaître les clefs de ses sessions passées. De la même manière, un ancien membre d'un groupe n'a droit qu'à la clef de sa session. Il ne doit pas accéder à toute communication après cette session. Pour voir l'intérêt de ces deux propriétés, revenons à notre exemple de la société vendant ses programmes sur internet. Il est tout à fait évident qu'un membre actuel du groupe d'abonnés ne peut en aucun cas regarder des programmes qui étaient diffusés avant la date d'abonnement de ce membre ou bien qui seraient diffusés après la date de fin de l'abonnement, vu qu'il n'a pas payé pour ces deux périodes.

Ces deux propriétés sont généralisées pour définir les propriétés de secret passé et futur qui traitent aussi le cas d'un intrus qui n'a jamais été membre du groupe. Nous notons ici que la notion de secret futur est à ne pas confondre avec la définition de propriété PFS donnée en Définition 2.4.1.6.

Définition 2.4.3.6 *Secret futur.*
Cette propriété garantit qu'un intrus passif connaissant un sous-ensemble d'anciennes clefs de groupe ne peut pas découvrir un ensemble ultérieur de clefs de groupe. En d'autres termes, aucun sous-ensemble de clefs anciennes de groupe ne peut être utilisé pour découvrir des clefs suivantes de ce groupe.

Définition 2.4.3.7 *Secret passé.*
Cette propriété garantit qu'un intrus passif connaissant un sous-ensemble de clefs de groupe ne peut pas découvrir les clefs précédentes de groupe. En d'autres termes, aucun sous-ensemble de clefs de groupe ne peut être utilisé pour découvrir les clefs antérieures de ce groupe.

Les deux propriétés de secret futur et de secret passé peuvent être combinées pour ne faire qu'une seule propriété : l'indépendance de clefs définie comme suit :

Définition 2.4.3.8 *Indépendance de clefs.*
Aucun sous-ensemble de clefs de groupe ne peut être utilisé pour la découverte d'aucun autre sous-ensemble.

La relation entre ces différentes propriétés est intuitive. En effet, les deux premières propriétés (secrets futur et passé faibles) sont différentes des autres dans le sens où, elles supposent que l'intrus est un membre courant ou ancien du groupe. Les autres propriétés incluent le compromis des clefs de groupe. Les propriétés de secret futur et secret passé sont plus fortes que celles de secret futur faible et secret passé faible. Ces deux propriétés englobent la propriété de secret de groupe et la propriété d'indépendance de clefs englobe le reste.

2.4.4 Résumé

Les protocoles de gestion de clefs ont généralement deux principaux objectifs. Le premier objectif est d'établir une clef du groupe entre un certain nombre de participants. Le deuxième objectif est de suivre l'évolution du groupe et maintenir la validité de la clef du groupe suite aux changements de la composition du groupe (par exemple l'ajout ou la sortie d'un ou plusieurs membres du groupe) ou bien la modification de la structure de ce groupe (par exemple, la montée ou la descente de grade ou de classe d'un ou plusieurs membres du groupe).

Les propriétés de sécurité que doit satisfaire un protocole de gestion de clefs doivent alors tenir compte de ces deux fonctionnalités. Nous distinguons donc deux grandes classes de propriétés : celles qui sont indépendantes de la dynamique du groupe dans le temps, et celles étroitement liées à cette dynamique. Un résumé des propriétés évoquées dans ce chapitre est donné par la Figure 2.6.

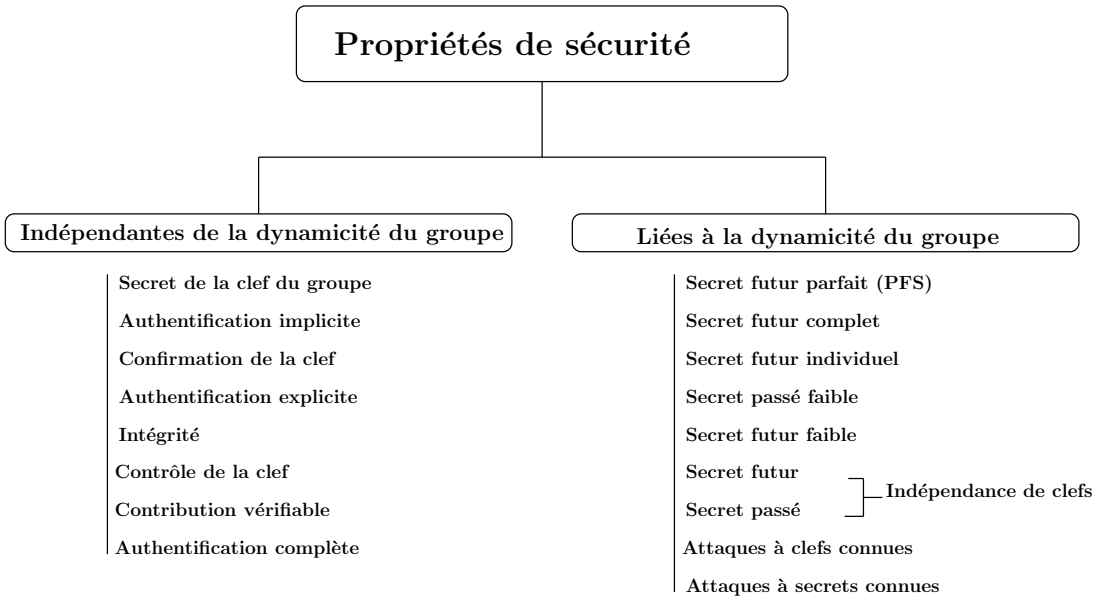


FIG. 2.6 – Résumé des propriétés des protocoles de gestion de clefs

Un résumé des propriétés indépendantes de la dynamique (respectivement, liées à la dynamique), ainsi que de leurs contextes de définition et leurs significations sont donnés en Table 2.1 (respectivement, en Table 2.2).

2.5 Impact des agents malhonnêtes

Une autre caractéristique assez intéressante différencie les protocoles de groupe de ceux à deux participants. Il s'agit de l'impact des participants malhonnêtes sur le fonctionnement du protocole. Un agent malhonnête est généralement une personne dont les clefs privées sont détenues par l'intrus. Cette idée est très générale puisqu'un agent malhonnête peut collaborer avec l'intrus en lui fournissant quelques informations mais pas toutes ses connaissances privées. Un agent malhonnête est défini comme une personne qui collabore avec l'intrus. Il y a trois niveaux de malhonnêteté :

– un participant fortement corrompu : il partage ses clefs privées avec l'intrus ;

Propriété	**Référence**	**Signification**
Secret de la clef du groupe	[55]	Il est impossible à un intrus passif de calculer la clef du groupe.
Authentification implicite	[67], [79]	Chaque membre du groupe est assuré qu'aucune personne extérieure ne connait la clef.
Confirmation de la clef	[67], [79]	Chaque membre est assuré que son groupe possède actuellement la clef du groupe.
Authentification explicite	[67]	Chaque participant identifié est connu pour avoir la clef du groupe établie. = authentification implicite + confirmation de la clef.
Intégrité	[79]	Définie pour les protocoles contribuants. Chaque partie est assurée que sa clef secrète est fonction seulement des contributions de tous les participants du groupe.
Contrôle de la clef	[67]	Aucune partie du groupe ne peut contrôler ou prévoir la valeur de cette clef.
Contribution vérifiable	[79]	Chaque partie est assurée de la contribution des autres membres du groupe concernant la clef à partager.
Authentification complète	[79]	Tous les participants calculent la même clef seulement si chaque participant contribue à cette clef.

TAB. 2.1 – Résumé des propriétés d'un groupe statique

- un participant faiblement corrompu : il fait signer et déchiffrer les messages que le protocole prévoit. Néanmoins, il est apte d'accepter des messages de l'intrus qui lui permet de tricher mais sans réagir directement d'une manière malhonnête.
- un participant accidentellement corrompu : le participant permet involontairement à l'intrus de surveiller le trafic entrant du réseau.

Dans un contexte de groupe à plusieurs participants, il y a un grand risque que certains membres soient corrompus. Les protocoles de groupe doivent alors tenir compte de ce grand risque et donc pouvoir prévenir les attaques de l'intérieur. Ce besoin est dû à plusieurs raisons. En effet :

- La possibilité d'avoir des attaques intérieures dans un contexte de groupe représente une différence qualitative du contexte de deux-parties où les attaques intérieures ne sont pas concernées.
- Quand quelqu'un de l'intérieur est malicieux, on ne peut pas lui interdire d'avoir la clef de session du groupe auquel il appartient. Néanmoins,
  - il ne doit pas avoir la possibilité d'avoir les clefs des groupes auxquels il n'appartient pas ;
  - il ne doit pas usurper l'identité d'un des autres agents honnêtes ;
  - il ne doit pas être capable de conduire les autres participants à calculer des clefs de session différentes sans s'en apercevoir.

Différents modèles de sécurité tel que [54, 15] ont vu le jour, ayant pour but de traiter les attaques par des participants malhonnêtes.

Propriété	**Référence**	**Signification**
Secret futur parfait	[67]	Le compromis des clefs à long terme n'entraîne pas le compromis des clefs de sessions passées.
Secret futur complet	[79]	Première variante de PFS sans aucune modification des messages des anciennes sessions du protocole.
Secret futur individuel	[79]	Deuxième variante de PFS avec manipulation de messages de quelques membres du groupe.
Attaques à clefs connues	[67]	Le compromis des *clefs de sessions passées* permet, soit à un intrus passif de compromettre des clefs de sessions futures, soit à un intrus actif d'espionner des sessions futures.
Attaques à secrets de session connus	[79]	Le compromis des *secrets de sessions passées* permet, soit à un intrus passif de compromettre des clefs de sessions futures, soit à un intrus actif d'espionner des sessions futures.
Secret passé faible	[55]	Les clefs passées du groupe ne doivent pas être découvertes par les nouveaux membres du groupe.
Secret futur faible	[55]	Les nouvelles clefs du groupe ne doivent pas être découvertes par les anciens membres du groupe.
Secret futur	[55]	Aucun sous-ensemble de clefs anciennes du groupe ne peut être utilisé pour découvrir les clefs suivantes de ce groupe.
Secret passé	[55]	Aucun sous-ensemble de clefs du groupe ne peut être utilisé pour découvrir les clés antérieures de ce groupe.
Indépendance de clefs	[55]	Regroupe le secret futur et le secret passé.

TAB. 2.2 – Résumé des propriétés d'un groupe dynamique

2.6 Conclusion

Nous avons détaillé dans ce chapitre les protocoles paramétrés. Ces protocoles peuvent avoir comme paramètre le nombre de participants impliqués. Ils peuvent aussi avoir des structures de données définies récursivement telles que les tables ou les listes dont la longueur est un paramètre de ces protocoles. Nous avons consacré plus d'intérêt pour les protocoles paramétrés par le nombre de participants vu qu'ils sont utilisés dans plusieurs applications. Nous avons également détaillé la gestion de clefs, un processus crucial pour la sécurisation des communications des protocoles de groupe. Ces protocoles sont conçus pour satisfaire des propriétés de sécurité bien particulières qui diffèrent de celles des protocoles à deux participants.

Le chapitre suivant sera consacré aux problèmes auxquels se confronte la vérification de tels protocoles. Nous détaillerons ensuite quelques travaux qui sont dédiés à leur vérification.

3

Vérification de protocoles de groupe

Sommaire

3.1 Introduction

La vérification des protocoles cryptographiques standards gérants deux ou trois participants [34] a donné beaucoup de résultats intéressants en se basant sur le modèle de Dolev Yao [45]. Dans les quelques années passées, d'intenses travaux de recherche basés sur ce modèle ont conduit au développement de plusieurs formalismes et outils de vérification des protocoles cryptographiques. Ce domaine de recherche a été donc bien exploité. De nos jours, d'autres types de protocoles encore plus complexes tels que les protocoles de groupe [65, 95] a suscité l'intérêt des travaux de recherche. En effet, ce genre de protocoles constitue un défit pour la vérification vu qu'il présente de nouvelles caractéristiques telles que l'implication d'un nombre arbitraire non borné de participants ou encore la considération d'autres propriétés de sécurité

que celles couramment utilisées pour les protocoles standards. Diverses méthodes d'analyse ont été menées pour vérifier ce type de protocoles. Les analyses varient entre celles qui cherchent à trouver des résultats de décidabilité pour ces protocoles [57, 59, 60, 99], et celles qui cherchent à les falsifier, i.e. trouver des attaques sur ces protocoles [73, 81, 94, 96]. Elles varient aussi entre celles effectuées manuellement (e.g. [81]) et celles effectuées par des outils automatiques ou semi-automatiques (e.g. [94]).

Ce chapitre a pour but de survoler les travaux effectués pour la vérification des protocoles de groupe. Comme nous avons détaillé les caractéristiques des protocoles de groupe au Chapitre 2, nous commençons en Section 3.2 par déduire les problèmes que peut rencontrer la vérification de tels protocoles. Nous présentons ensuite en Section 3.4 les travaux ayant abouti à des résultats théoriques. Dans cette catégorie, quelques résultats de décidabilité [57, 59, 60, 99] de certaines classes des protocoles de groupe ont vu le jour. Nous donnons en Section 3.3 quelques travaux ayant pour but de rechercher d'éventuelles attaques pour quelques protocoles de groupe et qui ont mené à des attaques intéressantes [81, 73, 96, 94]. Un corpus de protocoles de groupe et d'attaques sur ces protocoles peut être trouvé dans [3]. Finalement, nous présentons en Section 3.5 quelques travaux que nous avons effectué dans le but de retrouver des attaques sur quelques protocoles et d'en découvrir de nouvelles.

3.2 Problèmes liés à la vérification des protocoles de groupe

De nombreux travaux ont été effectués pour vérifier formellement les protocoles relativement classiques [5, 103]. Ces protocoles se caractérisent par la manipulation d'un nombre fixé de messages échangés au cours des sessions. Ces messages doivent aussi avoir une structure bien définie et fixée. Parmi les moyens de modélisation de ces protocoles, on peut par exemple exprimer leurs étapes comme étant de simples règles de réécriture. Grâce aux études de ces protocoles classiques, nous avons pu assister au développement de plusieurs outils automatiques de vérification de ces protocoles (par exemple [4]).

Cependant, dans [65], Meadows évoque d'autres protocoles plus compliqués présentant un défi pour la vérification. Il s'agit des protocoles de groupe dont le nombre de participants peut être arbitrairement grand. La vérification de ces protocoles est confrontée à plusieurs problèmes. En effet, comme nous venons de voir en Section 2.2.2, la sécurité des communications au sein des groupes n'est pas nécessairement une extension d'une communication sécurisée entre deux parties. Elle est beaucoup plus compliquée. Une fois la communication commencée, le groupe peut changer de structure en ajoutant ou en supprimant un ou plusieurs membres. Les services de sécurité sont alors plus élaborés. Ensuite, vu que les besoins en sécurité sont généralement liés aux protocoles, il est difficile de décrire les propriétés que le protocole doit satisfaire. Cette phase de spécification des propriétés est critique : toute erreur (de compréhension ou de formalisation de ces propriétés) peut engendrer de fausses failles de sécurité. En outre, la plupart des outils et des méthodes se sont concentrés seulement sur les propriétés de sécurité liées à l'atteignabilité telles que l'authentification ou le secret. Or, les protocoles de groupe considèrent des propriétés de sécurité plus développées (voir Section 2.4), liées à la dynamique du groupe. Par exemple, vérifier que, dans un protocole d'accord de clefs, tous les membres arrivent à déduire la même clef du groupe.

Au contraire des protocoles dits classiques, les protocoles de groupe mettent en cause un nombre de participants non borné. Même une exécution légale d'un de ces protocoles peut mener à un nombre non borné d'étapes, ce qui provoque une explosion du nombre d'états at-

teignables pour les techniques d'exploration d'états, souvent utilisées pour la vérification des protocoles cryptographiques. En outre, la plupart des approches automatisées de vérification de protocoles nécessitent un modèle concret. La taille du groupe doit alors être fixée à l'avance. Or, cette contrainte restreint considérablement le nombre d'attaques détectables.

La caractérisation des protocoles de groupe par le nombre non borné de participants fait apparaître de nouveaux défits à leur vérification. En effet, gérer un grand nombre de participants suppose implicitement qu'un ou plusieurs de ces participants ont à gérer un nombre non borné de messages échangés. Cette situation, illustrée par la Figure 3.1, représente le cas d'un groupe formé de deux parties : une composée d'une seule entité principale, et une deuxième partie composée du reste du groupe. Les messages sont échangés entre cette entité principale et chacun des autres participants de la deuxième partie. Les participants de la deuxième partie ne communiquent pas entre eux directement.

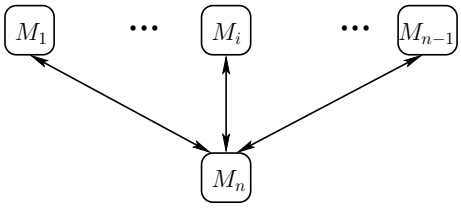


FIG. 3.1 – Un groupe structuré en deux parties

Comme exemple de protocoles présentant cette situation, nous pouvons citer les protocoles de distribution de clefs décrits en Section 2.3.1, mais aussi les protocoles d'accord de clefs décrits en Section 2.3.3. Nous sommes donc dans un contexte de groupe de plusieurs participants voulant générer une clef commune. En outre, il existe une entité appelée le serveur ou le leader qui a pour rôle de collecter toutes les contributions des différents participants.

La gestion d'un nombre non borné de messages peut aussi entraîner la nécessité de générer des messages à structure non bornée comme les listes et par la suite la nécessité de traitements itératifs de ces structures de données. Comme exemple de participants de protocole pouvant effectuer des traitements itératifs, nous trouvons le cas d'un serveur qui collecte les contributions des autres participants afin de générer la clef. Une fois, qu'il *a récupéré* les messages contenant les contributions, il construit la clef ou une information menant à la clef et *envoie* une *liste* de messages contenant cette information à tous les participants. Comme exemple de protocole de cette catégorie, nous pouvons citer le protocole de Asokan-Ginzboorg [6] qui suit la structure illustrée par la Figure 3.1. Cet aspect de traitement de listes est omniprésent dans la plupart des protocoles de groupe d'autant plus que ces listes nécessitent parfois un traitement récursif. Ce traitement a pour but de récupérer des informations des messages de la liste reçue afin de pouvoir générer le message que le participant veut envoyer. Ce genre de situation peut aussi exister dans le cas d'un groupe structuré sous forme de chaîne, i.e. les messages sont échangés d'un participant à un autre. Cette situation est illustrée par la Figure 3.2. Un membre de groupe reçoit un message de son voisin précédant (à gauche), ajoute son traitement à ce message et l'envoie à son prochain voisin (à droite). Finalement, quand le dernier participant reçoit le dernier message de cette chaîne, il génère le message à envoyer et le transmet à tous les participants.

Nous citons comme premier exemple pour cette situation, le protocole GDH (et les versions dérivées : A-GDH ...) du projet Cliques qui a pour objectif de générer une clef de groupe

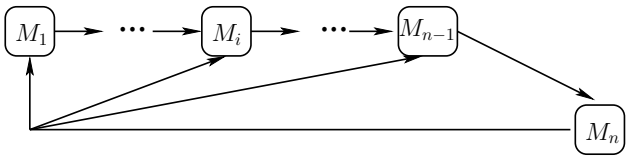


FIG. 3.2 – Un groupe structuré sous forme de chaîne

(voir Figure 2.5 de la Section 2.3.3). À la réception d'un message dans la phase de contribution, chaque membre traite une liste de messages en exponentiant chaque composant de cette liste par sa contribution personnelle à la clef du groupe. Le dernier membre reçoit une liste de messages. Il calcule sa clef à partir de la dernière composante de la liste reçue, exponentie chacune des composantes de cette liste par sa contribution à la clef du groupe, et envoie cette nouvelle liste au reste du groupe.

Comme deuxième exemple, nous citons le protocole RA (*Recursive Authentication*) [23] qui a pour but de permettre à un serveur de distribuer une clef de session pour n'importe quel nombre de membres. Nous nous focalisons dans ce protocole, sur la seule action récursive du serveur. Ce serveur reçoit une liste de paires de requêtes de participants qui veulent établir une clef de session entre eux. Cette liste de paires de requêtes est de taille variable et peut être large. Le serveur traite la liste d'une manière itérative, génère des clefs de session, une pour chaque paire de requêtes, et les distribue. Par exemple, le serveur peut recevoir un message m de la forme $m = h_{K_c}(C, S, N_c, h_{K_b}(B, C, N_b, h_{K_a}(A, B, N_a, -)))$ qui représente un ensemble de requêtes de paires de participants souhaitant l'obtention d'une clef de session. Dans ce message, N_a, N_b, N_c sont des nonces générées respectivement par A, B et C. Les clefs K_a, K_b, K_c sont des clefs à long terme partagées entre S et respectivement A, B et C. Le message $h_k(m)$ désigne le message $\langle hash(k, m), m\rangle$. La constante $-$ désigne la fin de la liste des requêtes et donc la fin du traitement récursif. Le serveur S doit traiter ce message d'une manière récursive afin de générer des certificats pour ces différents participants. Il génère tout d'abord deux certificats pour S : $\{K_{cs}, S, N_c\}_{K_c}$ et $\{K_{bc}, B, N_c\}_{K_c}$. Il génère par la suite deux certificats pour B : $\{K_{bc}, C, N_b\}_{K_b}$ et $\{K_{ab}, A, N_b\}_{K_b}$. Finalement, il génère un certificat pour A : $\{K_{ab}, B, N_a\}_{K_a}$.

En conclusion, la vérification des protocoles de groupe nécessite le traitement de certains calculs itératifs et récursifs, et donc fait partie de la vérification des protocoles dits *r*écursifs. Il s'agit des protocoles où les pas comprennent des calculs récursifs et/ou itératifs.
C'est le cas de la plupart des protocoles de groupe, que ce soient centralisés présentés en Section 2.3.1, ou distribués décrits en Section 2.3.3. En effet, pour le cas centralisé, on trouve généralement des calculs à la fois récursifs et itératifs puisque le serveur attend toutes les requêtes des différents participants et puis procède par calcul récursif afin de calculer la clef du groupe ou encore une information commune au groupe. Quant au cas distribué, ce traitement est le même mais il est nécessaire au niveau de chaque membre du groupe. Comme exemples de protocoles récursifs, on peut citer IKE, RA [23] ou encore GDH.2 [80].

La nécessité du traitement d'un nombre non borné de messages échangés ainsi que les calculs récursifs et itératifs effectués par les participants ont fait que la vérification des protocoles de groupe paraît difficile. Il s'agit donc d'un défit à relever par les méthodes de vérification de protocoles cryptographiques, surtout que ces méthodes vérifient des protocoles dont le nombre de messages dans une session est fixe et que la structure de ces différents messages est toujours fixe au départ et finie.

En résumé, la vérification des protocoles de groupe se confronte à plusieurs difficultés. Ces problèmes sont résumés en Figure 3.3.

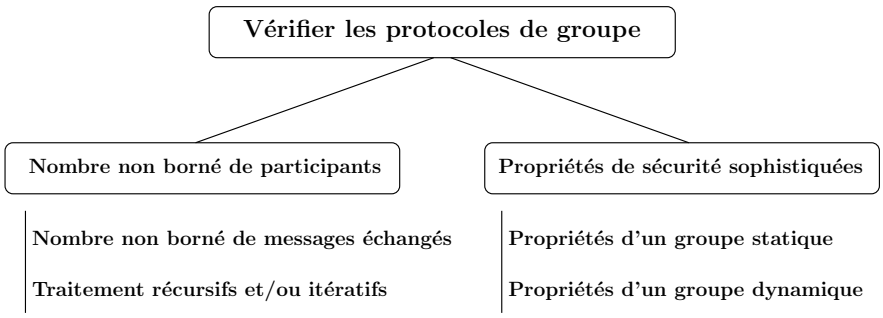


FIG. 3.3 – Problèmes liés à la vérification des protocoles de groupe

3.3 État de l'art : résultats expérimentaux

La vérification des protocoles de groupe, avec toutes les propriétés qui les caractérisent, a suscité l'intérêt de plusieurs travaux qui visaient à la recherche d'éventuelles attaques sur ces protocoles. Quelques travaux [77, 93, 94, 96] se sont basés sur des outils automatiques ou semi-automatiques qui étaient conçus pour la vérification des protocoles standards à un nombre de participants fixe mais qui nécessitent la plupart du temps des extensions afin de spécifier et de compiler les protocoles de groupe. Vu que les protocoles de groupe impliquent un nombre non borné arbitraire de participants, les méthodes se basant sur l'exploration des états possibles du protocole étudié ne peuvent pas, toutes seules, vérifier ce genre de protocoles puisque l'espace de recherche serait trop large. Quant aux prouveurs de théorèmes, bien que cela nécessite la preuve de lemmes d'une manière interactive et que cela demande beaucoup de temps pour arriver à un résultat, ils permettent de traiter directement les espaces infinis et donc de manipuler les protocoles d'une manière générale sans avoir à fixer le nombre de participants à l'avance. D'autres travaux [73, 81, 82] ont consisté en des analyses manuelles des protocoles de groupe. Ils ont pu trouver des attaques intéressantes sur ces protocoles, se basant sur des actions rusées de l'intrus.

Cette section a pour but de survoler les travaux effectués pour rechercher des attaques sur les protocoles de groupe. Nous commençons en Section 3.3.1 par les travaux manuels et nous décrivons quelques attaques trouvées. Nous présentons ensuite en Section 3.3.2 quelques travaux qui ont été effectués pour la vérification des protocoles de groupe. Ces travaux ont utilisé des outils automatiques ou semi-automatiques et ont mené à la découverte de quelques attaques peu connues.

3.3.1 Analyse manuelle

La vérification des protocoles de groupe a été étudié dans le projet Cliques [95], constitué de plusieurs protocoles basés sur Diffie-Hellman de groupe (e.g.GDH, A-GDH, SA-GDH...). Plusieurs méthodes d'analyse ont été appliquées à ces protocoles. Un résultat intéressant a été obtenu par Pereira et Quisquater [81] qui ont réussi à trouver plusieurs attaques concernant ces protocoles.

Leur analyse est manuelle. La méthode proposée consiste à convertir le problème de possession d'une information par l'intrus en un problème de résolution d'un système d'équations

linéaires. Cette méthode est dédiée aux protocoles de Cliques où les messages ainsi que la clef du groupe se basent sur les exponentiations. Les messages dans le modèle [81] sont donc constitués des éléments d'un groupe G d'ordre principal q. Ce groupe G a comme générateur α qui est partagé par tous les participants. Le modèle proposé manipule deux ensembles E et R. E désigne les nombres aléatoires r_i et les clefs à long terme k_{ij}. Quant à R, il désigne l'ensemble de relations entre les éléments de G. Un élément r de R représente la relation entre α^x et α^{rx} pour toute valeur de x. Formellement, R est défini par $R = \{\prod r_i e_i \prod k_{jl} e_{jl} | e_i, e_{jl} \in \mathbb{Z}\}$. Par exemple, pour des nombres aléatoires r_1 et r_2, et une clef à long terme k_{12}, $r_1 k_{12}$ est une relation entre α^{r_2} et $\alpha^{r_1 r_2 k_{12}}$.

Ce modèle présente deux restrictions. La première restriction exprime qu'un élément secret de G ne peut pas être calculé par combinaison des autres éléments de G, i.e. seulement la combinaison des éléments de E avec des éléments de G est permise. La deuxième restriction consiste au fait que ce modèle ne permet pas de capturer les relations entre plus de deux éléments de G.

Intrus

Le modèle de [81] assume l'hypothèse de Diffie Hellman parfait, i.e. un élément de G peut être calculé d'une seule manière : en exponentiant α par les nombres aléatoires et les clefs. Dans ce contexte, un intrus ne peut faire que deux opérations. Il peut exponentier un élément de G par un élément de E. Il a aussi le droit d'obtenir de nouveaux éléments de G grâce aux **services disponibles**. Un service disponible est défini par la fonction s de G dans G qui à α^x retourne $\alpha^{p.x}$. À ce service correspond le poids $p \in S$ avec S est l'ensemble appelé ensemble des services disponibles. Concrètement, un service disponible est récupéré par l'intrus en envoyant un message α^x à un participant pour recevoir un message exponentié par le service p de ce participant, i.e. $\alpha^{p.x}$. Dans ce cadre, les opérations que peut faire l'intrus (citées ci-dessus) sont exprimées comme suit :

$$\begin{array}{lllll} (1) & Si\ e \in E_I\ et\ r \in R_I & alors & r.e \in R_I\ et\ r.e^{-1} \in R_I \\ (2) & Si\ p \in S\ et\ r \in R_I & alors & r.p \in R_I\ et\ r.p^{-1} \in R_I \end{array}$$

avec $E_I \subseteq E$, $R_I \subseteq R$ et S l'ensemble des services disponibles.

Méthode

D'après la méthode [81], les sessions du protocole considérées ainsi que les rôles de chaque participant sont tout d'abord fixées. Cette méthode est composée de trois étapes. La **première étape** consiste à définir, pour l'exécution considérée, trois ensembles S, E_I et R_S tels que :

- S désigne l'ensemble de services disponibles ;
- E_I désigne l'ensemble des connaissances (éléments atomiques : nombres aléatoires et clefs) initiales de l'intrus ;
- R_S désigne l'ensemble de secrets qui ont la forme de relations tel qu'il est défini précédemment pour R.

Par exemple, le protocole A-GDH.2 à quatre participants P_1, P_2, P_3 et P_4 est décrit par la Figure 3.4. En supposant que l'intrus connaît r_3, le protocole décrit par la Figure 3.4 peut être décrit par le système suivant :

$$\begin{array}{lcl} S & = & \{r_1, r_2, r_3, r_4 k_{14}, r_4 k_{24}, r_4 k_{34}\} \\ E_I & = & \{r_3\} \\ R_S & = & \{r_1 k_{14}^{-1}, r_2 k_{24}^{-1}, r_3 k_{34}^{-1}, r_4\} \end{array}$$

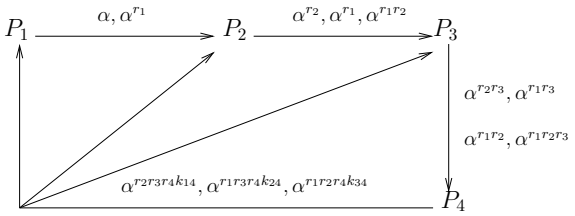


FIG. 3.4 – A-GDH.2 à quatre participants

La **deuxième étape** consiste à éliminer tout élément correspondant à E_I des expressions de R_S et de S, si les conditions décrites dans [81] sont satisfaites. Comme résultat de cette étape, on obtient deux autres ensembles R^1 et S^1. Dans le cas de l'exemple de départ, nous obtenons :

$$\begin{aligned} S^1 &= \{r_1, r_2, r_4 k_{14}, r_4 k_{24}, r_4 k_{34}\} \\ R^1 &= \{r_1 k_{14}^{-1}, r_2 k_{24}^{-1}, k_{34}^{-1}, r_4\} \end{aligned}$$

L'objectif de la **troisième étape** est de savoir si un élément de R^1 peut être obtenu de S^1 par exploitation de la règle (2) de l'intrus défini précédemment. D'où, la construction d'un système linéaire dont on va détailler la construction. Le nombre de variables de ce système correspond au nombre d'éléments de S. En effet, le système contient une variable par service disponible (et donc par élément de S). Ensuite, le nombre d'équations du système linéaire correspond au nombre d'éléments de E, i.e. au nombre des nombres aléatoires et des clefs à long terme. En effet, le système contient une équation par élément de E. Par exemple, le système linéaire correspondant à l'exemple expliqué relatif à S^1 et R^1, comprend six équations respectivement, pour r_1, r_2, r_4, k_{14}, k_{24} et k_{34}. Pour chacune de ses équations, sa partie droite correspond au poids de l'élément de E dans le secret étudié. Par exemple, dans le même exemple relatif à S^1 et R^1, en considérant le secret $r_1 k_{14}^{-1}$, la valeur droite de l'équation correspond à l'élément r_1 est 1. Quant à la partie gauche d'une équation correspondante à un élément e de E, le coefficient de chacune des variables du système linéaire correspond au poids de e dans l'élément de S correspondant à la variable considérée. Par exemple, pour le système linéaire correspondant à S^1 et R^1, en considérant l'équation relative à r_4, le coefficient des variables x_3 (qui correspond à $r_4 k_{14}$), x_4 (qui correspond à $r_4 k_{24}$) et x_5 (qui correspond à $r_4 k_{34}$) est de 1.

Le système linéaire correspondant à l'exemple correspondant à R^1 et S^1, en considérant le secret de $r_1 k_{14}^{-1}$, est représenté de la manière suivante :

$$\begin{array}{c} \\ r_1 \\ r_2 \\ r_4 \\ k_{14} \\ k_{24} \\ k_{34} \end{array} \begin{array}{c} \begin{array}{ccccc} r_1 & r_2 & r_4 k_{14} & r_4 k_{24} & r_4 k_{34} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{bmatrix} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

Le système linéaire est par la suite résolu. Si le système est inconsistant (c'est le cas de cet exemple), alors la propriété est vérifiée. Sinon, il existe une attaque pour le protocole considéré et il reste donc de la retrouver à partir de la solution du système linéaire.

Résultats

Grâce à cette approche, Pereira et Quisquater ont pu trouver des faiblesses dans plusieurs protocoles de CLIQUES. Les protocoles analysés dans [81] sont les protocoles A-GDH.2, A-GDH.2-MA et SA-GDH.2. Pour le protocole A-GDH.2, les attaques trouvées concernent les propriétés de sécurité suivantes : l'authentification implicite, le secret futur parfait avec ses deux variantes, i.e. le secret futur individuel et le secret futur complet, et la résistance aux attaques à clefs connues. Par exemple, pour étudier la propriété de secret futur complet pour le protocole A-GDH.2, les auteurs supposent qu'une session du protocole a été exécutée et que l'intrus connaît toutes les clefs à long terme de cette session. Ainsi, $E = \{k_{14}, k_{24}, k_{34}\}$. Concernant le secret, vu que l'intrus ne peut pas influencer de sessions antérieures, la clef du groupe calculée par les différents participants n'est pas le résultat de la mise en exposant de messages, mais admet plutôt une seule valeur $\alpha r_1 r_2 r_3 r_4$. Ainsi, $R = \{r_1 r_2 r_3 r_4\}$. Quant à la propriété de secret futur individuel, on considère le même ensemble des connaissances de l'intrus E. Cependant, vu que l'intrus peut manipuler les sessions précédentes, l'ensemble de secrets est $R = \{r_1 k_{14}^{-1}, r_2 k_{24}^{-1}, r_3 k_{34}^{-1}, r_4\}$. Avec ce système, les auteurs ont pu trouver une attaque contre le secret futur individuel illustrée par la Figure 3.5. D'après la Figure 3.5, le participant P_1 calcule comme clef de groupe la clef

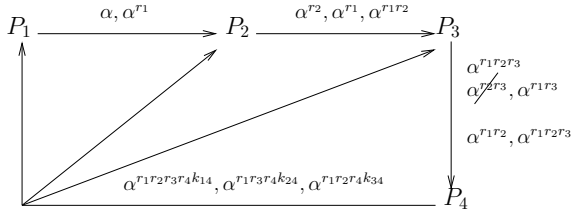


FIG. 3.5 – Attaque de secret futur individuel sur A-GDH.2 à quatre participants

$\alpha^{r_1 r_1 r_2 r_3 r_4}$ mais les autres participants continuent à calculer la même clef de groupe $\alpha^{r_1 r_2 r_3 r_4}$. En outre, si la clef k_{14} est compromise (ce qui est le cas ici) alors l'intrus est capable de calculer la même clef que les autres membres à partir du message envoyé par P_4.

Résultat générique d'insécurité

Les mêmes auteurs étudient dans [82] une famille de protocoles d'accord de clefs de groupe authentifié qui est définie sur une généralisation du protocole GDH du projet Cliques. Le modèle utilisé se base sur le modèle de *strand space* [98]. Un *strand* est une séquence de nœuds représentant la vue de l'exécution du protocole d'un certain participant. Les termes associés à ces nœuds sont des termes avec des signes + ou − pour désigner respectivement un terme envoyé et un terme reçu. Un *bundle* est un graphe direct dont les arêtes expriment les relations de causalité entre les nœuds. Un symbole $\rightarrow$ connecte deux nœuds dont les termes associés sont de la forme $+t$ et $-t$. Un symbole $\Rightarrow$ connecte deux nœuds du même *strand*.

Le protocole GDH permet à un ensemble M de n participants $M = \{M_1, .., M_n\}$ de partager une clef $\alpha^{r_1,..,r_n}$. Une exécution régulière de ce protocole peut être décrite par deux éléments. Le premier élément est un *bundle* contenant n *strands* qui correspondent, chacun à un participant actif M_i. Cet élément exprime comment les messages sont échangés. Le deuxième élément est

un ensemble de n chemins appelés historiques et qui expriment comment les termes GDH sont calculés. Cet ensemble de chemins admet plusieurs caractéristiques telles que le calcul du terme envoyé par un participant à partir du terme qu'il a reçu ou encore le calcul de la clef du groupe à partir du dernier message reçu. Le détail de ces caractéristiques est dans [82]. Des propriétés de la famille des protocoles GDH sont présentées ainsi que leurs preuves. Parmi ces propriétés, nous trouvons par exemple le fait que la contribution, en tant que nombre aléatoire, d'un participant M_j à la valeur finale d'un participant M_i (la valeur que va utiliser M_i pour le calcul de la clef) est toujours r_j. À partir de ces propriétés, les auteurs prouvent que, pour $n \geq 3$ (un protocole impliquant plus que 3 membres), le produit $R_i.K_i$ que le participant M_i utilise pour calculer la clef du groupe peut être écrit comme étant le produit des contributions (contribution de M_j à M_k qui sont différents de l'intrus) et des clefs connues par l'intrus. Cette propriété présente une première étape pour prouver un résultat générique d'insécurité citant qu'il est impossible d'écrire un protocole appartenant à la famille des protocoles GDH satisfaisant la propriété d'authentification implicite de la clef pour un groupe qui comprend au moins quatre membres. Dans ce contexte, l'intrus sélectionne trois membres M_i, M_j et M_k parmi l'ensemble des participants M. Il sélectionne aussi deux ensembles disjoints de participants S_j et S_k tels que $M_k \in S_j$, $M_j \in S_k$, $M_i \notin S_j \cup S_k$ et $S_j \cup S_k \cup \{M_i\} = M$. Cette sélection doit respecter certaines caractéristiques surtout à propos d'une valeur p définie dans [82] et qui va correspondre à $R_i.K_i$. Avec ces sélections, l'intrus est capable de construire une paire (g_1, g_2) avec $g_2 = g_1^p$ selon un algorithme proposé dans [82]. L'intrus remplace par g_1 la valeur qu'un participant M_i va utiliser pour déduire la clef du groupe. Le participant M_i va donc calculer comme clef de groupe $g_1^{R_i.K_i}$ qui n'est autre que g_2. Ainsi, la propriété d'authentification implicite de la clef n'est pas satisfaite pour un protocole GDH impliquant un nombre de participants dépassant trois ($n \geq 4$).

Autres études à la main des protocoles de groupe

Nam, Kim et Won [73] ont essayé d'analyser le protocole GKE.Setup : un des protocoles formant l'arrangement de clefs proposés par Bresson et al. [22]. Cet arrangement est constitué de trois protocoles : *GKE.Setup*, *GKE.Remove* et *GKE.Join*. Le protocole GKE.Setup permet à un ensemble d'utilisateurs mobiles de parvenir à une clef de session entre eux. Les deux autres protocoles permettent de traiter le dynamisme du groupe suite aux opérations d'addition (*GKE.Join*) et de sortie de membres (*GKE.Remove*). L'analyse effectuée considère trois propriétés de sécurité : l'authentification implicite de la clef, le secret futur et la résistance aux attaques à clefs connues. Trois attaques correspondant à ces propriétés ont été trouvées, impliquant un intrus actif. Les auteurs proposent par la suite une amélioration du protocole analysé pour satisfaire ces différentes propriétés.

3.3.2 Analyse automatique ou semi-automatique

Nous nous intéressons dans cette section aux attaques qui ont été trouvées pour les protocoles de groupe par des outils automatiques ou semi-automatiques. Nous commençons tout d'abord par le premier travail [77] qui a considéré les protocoles récursifs. Nous présentons ensuite l'extension de l'outil NRL faite par Meadows [66] pour spécifier et compiler les protocoles de groupe.

Le paragraphe suivant est dédié au travail effectué par Taghdiri et Jackson [96] pour analyser le protocole de Tanaka-Sato et l'attaque trouvée. Puis, nous nous intéressons à l'analyse faite par Steel [93, 94], utilisant l'outil Coral, des protocoles Asokan-Ginzboorg, Iolus et Tanaka-sato. Finalement, nous présentons le travail d'extension du langage de spécification CAPSL en

MuCAPSL [68] afin de pouvoir traiter les protocoles de groupe.

Première analyse des protocoles récursifs

L'analyse formelle de protocoles récursifs remonte à Paulson [77]. Il a étudié le protocole récursif d'authentification RA [23], proposé par Bull et Otway, faisant partie de son développement de la méthode inductive [78].

Paulson a pu modéliser ce protocole inductivement, et prouver ses propriétés de sécurité d'une manière générale en considérant des groupes de taille arbitraire. La méthode de Paulson utilise un langage à base de logique typée d'ordre supérieur. Les propriétés de sécurité sont prouvées par induction sur des traces qui consistent en une liste d'évènements (envoi des messages par les participants), en utilisant le prouveur interactif de théorèmes Isabelle/HOL [76]. L'intérêt de la méthode inductive est qu'elle traite directement le modèle à états infinis et qu'elle suppose l'implication d'un nombre arbitraire de participants permettant ainsi la preuve des propriétés des protocoles de groupe. Néanmoins, ces preuves ne sont pas faciles à mener et demandent du temps (des jours voire des semaines) pour être accomplies. En outre, si le protocole est défectueux, la méthode de Paulson ne fournit aucun support automatisé pour trouver l'attaque.

Extension de l'outil NRL

Meadows [66] a étendu l'analyseur de protocoles NRL afin de fournir une spécification formelle de la suite de protocoles GDOI (*Group Domain Of Interpretation*), proposée comme un standard IETF. Le but de cette suite de protocoles est de permettre à un contrôleur de groupe (ou serveur de clefs) de distribuer des clefs aux membres du groupe. L'analyse du [66] avait pour but de développer un ensemble de conditions (requirements) et de fournir une analyse formelle pour le protocole étudié. L'outil NRL est un outil de vérification qui combine les techniques de preuves de théorèmes et de model checking afin d'établir les propriétés de secret et d'authentification. L'intrus considéré est celui de Dolev Yao. Leur modèle considère aussi le cas des agents malhonnêtes. Il s'agit des agents qui partagent leurs clefs et d'autres informations privées avec l'intrus. Pour vérifier un protocole, sa spécification ainsi qu'une description des états qui correspondent à des situations d'attaques, sont données comme entrée à l'outil NRL. L'analyse du protocole se fait en arrière, i.e. en commençant par ces états d'attaques en appliquant les actions du protocole afin d'atteindre un état initial.

Dans l'étude faite sur GDOI [66], les serveurs de clefs sont supposés être honnêtes. Certains autres participants peuvent être malhonnêtes et dans ce cas, ils partagent toutes leurs informations avec l'intrus. Ce travail consiste en une définition des conditions (requirements) dans le contexte de GDOI mais qui peuvent être généralisées pour les protocoles de distribution de clefs. Quelques conditions sont relatives à la propriété d'authentification par rapport à un simple membre du groupe et par rapport à un contrôleur de groupe. Par exemple, un membre du groupe voudrait connaître que, s'il accepte une clef, alors cette clef a été générée par le contrôleur de son groupe. De même, un contrôleur de groupe voudrait connaître que, s'il a envoyé une clef à un membre du groupe, alors ce membre a demandé cette clef. D'autres conditions ont été définies pour GDOI telles que la condition de fraîcheur : si un participant reçoit une clef alors elle doit être la clef courante, relative à un certain point de temps par rapport à l'horloge de ce participant. En ce qui concerne le secret, mis à part le secret de la clef du groupe, exprimant que seuls les membres du groupe connaissent cette clef, ils considèrent aussi le secret futur et le secret passé. Ces deux propriétés précisent que les nouveaux membres du groupe ne connaissent pas les clefs anciennes et que les anciens membres ne doivent pas connaître les clefs futures.

L'outil NRL étendu a été aussi utilisé pour l'analyse de la suite de protocoles Cliques [65].

La considération de ces protocoles comme étude de cas, a nécessité le besoin d'étendre le modèle utilisé afin de tenir compte des opérations de Diffie-Hellman et de l'exponentiation. L'approche par NRL a été étendu afin de rendre possible la spécification et la compilation de cette suite de protocoles. Cependant, aucune attaque comme les attaques de Pereira et Quisquater [81] n'a été découverte. Ces attaques se basent sur le fait que, pour mener ces attaques, l'intrus procède par des actions complexes comme détaillé en Section 3.3.1.

Modélisation des protocoles de groupe par Taghdiri et Jackson

Taghdiri et Jackson [96] ont analysé un protocole de gestion de clefs en multicast proposé par Tanaka et Sato [97]. Ce protocole a été conçu pour un scénario où le groupe est très dynamique : les opérations d'addition (*join*) et de suppression (*leave*) de membres sont très fréquentes. Le but de la conception de ce protocole était donc de minimiser le nombre de mises à jour de la clef en fournissant sur demande des clefs aux agents. Le langage de modélisation utilisé dans [96] est le langage de spécification Alloy [53]. Cette modélisation est par la suite transmise à l'outil de vérification SAT de Alloy. Pour les propriétés de sécurité étudiées, il faut alors chercher des contre-exemples qui correspondraient à des attaques réelles sur le protocole considéré. Le modèle [96] analyse les protocoles sans fixer par avance le nombre de participants impliqués. Cependant, il ne modélise pas le cas d'intrus essayant de manipuler les messages du protocole afin de nuire à son fonctionnement.

La version du protocole de [97] a été analysée pour différentes propriétés de sécurité. La première propriété étudiée est celle représentée par une assertion exprimant que, aucune personne qui ne fait pas partie du groupe ne peut recevoir un message envoyé de l'intérieur du groupe. Cette assertion regroupe à la fois le fait qu'un nouveau membre de groupe ne peut accéder aux communications précédentes, et qu'un ancien membre ne peut accéder aux communications futures (propriétés connues sous le nom de secret passé et futur). Aucun contre-exemple n'a été trouvé pour cette propriété, ce qui prouve qu'elle est vérifiée pour le protocole analysé.

Une deuxième assertion est analysée. Elle exprime qu'aucun message, envoyé de l'extérieur du groupe, ne peut être reçu de l'intérieur du groupe. Cette assertion précise qu'un message envoyé de l'extérieur ne peut pas être interprété comme un message valide du groupe. Deux attaques ont été trouvées pour cette assertion. Dans l'une de ces deux attaques, deux participants m_1 et m_2 rejoignent le groupe en même temps. Ils obtiennent donc la même clef du groupe k. Le membre m_2 quitte le groupe mais envoie un message encrypté par k. Quand m_1 reçoit ce message, elle ne contacte pas le serveur pour obtenir la clef du groupe vu que le message reçu est encypté par sa propre clef k. Il accepte donc le message reçu sans penser que ce message provient de l'extérieur.

Une autre assertion a été étudiée exprimant le fait qu'un membre actuel du groupe doit être capable de décrypter tout message envoyé de l'intérieur du groupe. Un contre-exemple a été trouvé pour cette assertion. Dans le scénario correspondant, un membre m_1 envoie un message m au groupe composé de m_0 et m_1. Néanmoins, seul m_1 a la dernière version de la clef du groupe, et peut donc décrypter le message m. Ensuite, m_0 quitte le groupe tout en ayant la version antérieure (non valide) de la clef du groupe. Ainsi, même s'il était membre du groupe au moment de l'envoi du message m, il ne peut pas le recevoir (puisqu'il n'a pas la clef de décryption).

Dans [96], Taghdiri et Jackson ont proposé une version améliorée du protocole qui fera l'objet d'une analyse dans le paragraphe suivant où d'autres attaques de cette version améliorée sont trouvées.

Modélisation des protocoles de groupe par Coral

Coral [94] emploie le modèle inductif de Paulson en utilisant la logique du premier ordre, permettant de traiter des protocoles à un nombre non borné non fixé en avance de participants qui jouent des rôles en utilisant un nombre arbitraire de nonces frais et de clefs. Coral a été conçu pour automatiser la réfutation des conjectures inductives incorrectes dans le contexte de l'analyse des protocoles. Coral représente une implantation de la méthode de Comon-Nieuwenhuis [36] de preuve par consistance au niveau du prouveur de théorème de premier ordre SPASS [103]. [36] a montré que toute conjecture incorrecte sera réfutée en un temps fini. En outre, si une réfutation est trouvée, Coral fournit un contre-exemple en donnant la trace correspondante afin d'exploiter l'attaque sur le protocole.

Dans le modèle de [94], un protocole est modélisé comme étant l'ensemble de toutes les traces possibles. Une trace de messages est modélisée sous forme de liste représentant une séquence possible de messages envoyés par tout nombre de participants honnêtes suivant la spécification du protocole. Elle modélise aussi les messages composés par l'intrus. L'état de tout le système représentant le protocole est codé dans la trace, i.e. les connaissances des participants et l'intrus sont codés ensemble. Ceci permet d'ajouter des nombres non bornés de messages à la trace sous une seule règle de premier ordre, permettant ainsi la modélisation par exemple du protocole de groupe Asokan-Ginzboorg sans avoir à prédéterminer la taille du groupe.

Intrus. L'intrus considéré est celui de Dolev Yao avec une optimisation par rapport aux méthodes inductives : l'intrus envoie tout message qu'il peut composer à condition que ce message correspond à la forme d'un des messages du protocole. En effet, l'intrus ne va rien gagner en envoyant un message auquel un participant ne va pas répondre.

Modélisation du protocole. Pour un protocole, le nombre d'axiomes correspond au nombre de messages. Chaque axiome a cette interprétation : si xt est une trace contenant un message n adressé à un agent xa, alors xt peut être étendu par 'xa répond par le message $n + 1$'. Une fois que les axiomes du protocole sont écrits, les clauses modélisant la réception et l'envoi de ces messages par l'intrus sont générées automatiquement. Ces deux ensembles sont alors ajoutés à l'ensemble standard des axiomes décrivant les types, l'intrus...

Résultats. L'utilisation d'un modèle inductif a permis la spécification du protocole Asokan-Ginzboorg [6] pour un groupe de taille non fixée, ce qui a mené à la découverte de différentes attaques sur ce protocole. Le but du protocole Asokan est de construire une clef commune entre plusieurs participants. Cette clef est construite à partir des contributions de tous les membres du groupe. Des contre-exemples sont donnés avec des tailles de groupe deux puis trois. Par exemple, dans l'attaque à deux participants M_1 et M_2, le membre M_2 va finalement déduire une clef qui ne contient pas des informations de M_1. Ainsi, le membre M_1 n'a pas réellement contribué à la clef construite par M_2. Dans l'attaque à trois participants, l'intrus fait partie du groupe. Il a donc le droit de connaître la clef du groupe. Néanmoins, il réussit à diviser le groupe en deux parties où chaque partie détient une clef différente de l'autre. L'intrus, lui, connaît les clefs déduites des deux parties.

Coral a été aussi utilisé dans [93] pour modéliser et vérifier les protocoles de groupe assez complexes. En effet, la version améliorée du protocole de gestion de clefs de groupe multicast de Tanaka et Sato [97] a été analysé dans [93]. Deux attaques intéressantes ont été découvertes sur ce protocole. La première attaque consiste en un rejeu d'un message de mise à jour reçu par l'intrus antérieurement. Ce rejeu a pour conséquence de faire accepter une clef de groupe à un membre du groupe de telle façon que cette clef est déjà connue par l'intrus lorsqu'il était dans

le groupe. La deuxième attaque est similaire à la première utilisant aussi un rejeu d'un message fourni par le serveur à l'intrus lorsqu'il était membre du groupe et qui contient la clef du groupe. Après la demande d'un membre de la clef de groupe auprès du serveur, l'intrus se fait passer pour le serveur en envoyant le message qu'il a déjà récupéré avant et dont il connaît la clef de groupe correspondante.
En outre, ce même travail [93] a trouvé une attaque sur le protocole de gestion de clefs Iolus [71]. Ce protocole est constitué de trois sous-protocoles : *join*, *leave* et *send*. Nous nous intéressons plus précisément aux deux sous-protocoles : *join* et *leave* qui sont décrits comme suit :

Join			*Leave*		
1.	$M_i \rightarrow S$	$: \{Join\}_{K_{M_i}}$	1.	$M_i \rightarrow S$	$: \{Leave\}_{Ik_{M_i}}$
2.	$S \rightarrow M_i$	$: \{Ik_{M_i}, Gk_{n'}\}_{K_{M_i}}$	2.	$S \rightarrow All$	$: \{Gk_{n'}\}_{Ik_{M_j}} \forall j \neq i,\ M_j \in Group$
3.	$S \rightarrow All$	$: \{Gk_{n'}\}_{Gk_n}$			

Pour le sous-protocole Join, suite à la requête du participant M_i, le serveur S génère une nouvelle clef du groupe $Gk_{n'}$ qu'il va communiquer à tous les membres actuels du groupe, encryptée par l'ancienne clef du groupe Gk_n. Le nouveau membre va recevoir cette clef avec une nouvelle clef individuelle Ik_{M_i} qui sert pour sa communication avec le serveur. Concernant le protocole Leave, à la suite d'une requête de sortie d'un M_i, le serveur génère une nouvelle clef et la communique au reste du groupe mais cette fois en l'encryptant par la clef individuelle de chaque membre de ce groupe restant par peur que le membre sorti ne sache la nouvelle clef.

La propriété vérifiée pour ce protocole est celle vérifiant que, quand une clef Gk vient d'être mise à jour et d'être acceptée par un membre du groupe auquel l'intrus ne fait pas partie, l'intrus ne doit pas connaître cette clef. Cette propriété est ajoutée au système en tant que conjecture négative, exprimant que, pour qu'un protocole soit sûr, aucune trace où l'intrus connaît une clef acceptée par un membre du groupe, ne doit exister quand l'intrus se trouve à l'extérieur du groupe. L'attaque trouvée sur Iolus est une attaque de rejeu. Après l'ajout d'un premier participant a au groupe, l'intrus I envoie une requête join et il adhère au groupe. Un autre membre b rejoint le groupe. Le participant a veut maintenant quitter le groupe. Le serveur génère donc une nouvelle clef de groupe Gk' et l'envoie à b et à l'intrus en envoyant le message $m = \{Gk'\}_{Ik_b}, \{Gk'\}_{Ik_I}$. Ensuite, l'intrus quitte le groupe. Le serveur renouvelle alors la clef du groupe (Gk"). Cependant, l'intrus envoie le message m qu'il a reçu dernièrement et la clef du groupe est maintenant Gk' qui est connue par l'intrus bien qu'il ne fasse plus partie du groupe.

Analyse des protocoles de groupes par model checking

Le langage de spécification de protocoles CAPSL [43] a été étendu en MuCAPSL [68] pour pouvoir modéliser les protocoles de groupe. Le nouveau langage utilise, comme l'ancien, des rôles séparés pour spécifier un protocole. Ce langage se base sur d'autres types pour tenir compte des structures de groupe. Par exemple, mis à par le type *Participant* qui désigne un participant du protocole, un autre type a été ajouté (*GroupMember*) pour désigner un membre de groupe comportant deux informations : le participant qui contrôle le groupe (le propriétaire du groupe) et l'identifiant du groupe auquel appartient ce membre. En outre, pour tenir compte du dynamisme du groupe, d'autres attributs ont été définis. Par exemple, un attribut qui permet de sauvegarder la taille et la clef actuelle du groupe pour chaque membre du groupe. Les propriétés que peut considérer le langage MuCAPSL sont le secret et l'authentification sous deux formes permettant de prévenir un membre de groupe d'utiliser une clef ou un message qui n'a pas été envoyé par une autre partie. La spécification MuCAPSL sera traduite en une spécification en langage intermédiaire MuCIL. Dans ce langage intermédiaire, un protocole est modélisé comme

étant un système où l'envoi ou la réception d'un message entraîne une transition d'état. Les transitions sont exprimées par des règles de réécriture. Le langage MuCIL forme l'entrée de tout outil analysant des systèmes de transitions finis, modulo quelques petites modifications.

La suite des protocoles GDH.2 a été étudiée par cette méthode en utilisant l'outil Maude [2]. Les déclarations des types de MuCAPSL et les règles de l'intrus, lui permettant de dupliquer, supprimer ou encore rejouer des messages, ont été ajoutées aux règles dans Maude. La spécification en langage intermédiaire MuCIL du protocole GDH.2 a été exécuté par l'interpréteur Maude. À cette fin, les scénarii initiaux adoptés pour analyser le protocole GDH.2 considèrent un nombre borné de participants. La propriété considérée exprime le fait que tous les membres du groupe arrivent à partager la clef du groupe. Une attaque a été trouvée. Elle énonce qu'au moins deux membres du groupe finissent leur rôle dans le protocole mais dérivent deux clefs distinctes.

3.4 État de l'art : résultats théoriques

Plusieurs travaux ont considéré la vérification des protocoles récursifs. Nous rappelons que les protocoles récursifs sont des protocoles où les pas comprennent des calculs récursifs et/ou itératifs. C'est le cas de la plupart des protocoles de groupe que ce soient ceux qui sont centralisés (il existe une entité centrale : serveur ou leader qui contrôle le groupe) ou encore ceux qui sont distribués (il n'existe pas d'entité centrale qui choisit la clef du groupe). Pour le cas centralisé, on trouve généralement des calculs à la fois récursifs et itératifs puisque le serveur attend toutes les requêtes des différents participants (d'où l'action itérative) et puis procède par calcul récursif afin de calculer la clef du groupe ou encore une information commune au groupe. Quant au cas distribué, ce traitement est le même mais c'est au niveau de chaque membre du groupe. Comme exemples de protocoles récursifs, nous pouvons citer RA [23] ou A-GDH.2 [80]. Nous donnons dans ce qui suit les différents travaux théoriques effectués dans le cadre des protocoles récursifs.

Nous commençons en Section 3.4.1 par présenter le travail fait par Küsters [60] en 2004 qui a utilisé les automates d'arbres pour représenter les protocoles récursifs. Nous présentons après en Section 3.4.3 le travail de Truderung en 2005 qui se base sur une classe de clauses de Horn. Puis, nous considérons deux extensions de ce travail en Section 3.4.3 : une première extension faite par Truderung et Küsters [59] en 2007, et une deuxième extension faite par Kurtz [58] en 2007. Finalement, nous donnons en Section 3.4.4 le plus récent travail, à notre connaissance, qui porte sur la validation des protocoles de groupe par le biais des automates d'arbres dans le cas d'un intrus passif.

3.4.1 Automates d'arbres pour les protocoles récursifs (Küsters)

[60] introduit les automates d'arbres (TTAC) afin de modéliser la récursion et permettre aux participants du groupe de construire des messages structurés. Les automates d'arbres sont utilisés dans ce modèle pour deux caractéristiques. En effet, ils permettent tout d'abord l'application de tout un ensemble de règles de réécriture récursivement à un terme. En outre, Ils permettent de générer de nouvelles constantes. Dans ce qui suit, nous présentons le modèle utilisé ainsi qu'un résumé des résultats obtenus et de quelques limites de ce modèle.

Modèle

Les messages dans ce modèle sont construites sur une signature contenant les opérateurs classiques sans opérateurs algébriques. Plus précisément, la signature considérée comprend les variables, les constantes et les symboles de fonctions tels que la fonction de hachage, les paires,

les encryptions symétriques et asymétriques. Les clefs utilisées sont des clefs atomiques. Cette signature est étendue par un ensemble non borné de constantes appelées constantes anonymes. Cependant, ces constantes anonymes ne sont pas à utiliser en position de clefs.

Protocole.
Un protocole est un tuple de famille finie de participants et un ensemble fini de connaissances initiales de l'intrus. Un participant est défini comme une séquence finie d'actions *r*eceive-send. Une action *receive-send* n'est plus considérée comme une simple règle de réécriture mais elle est plutôt représentée par un automate d'arbre (*t*ransducer). La dernière action de ce transducer est une action de sortie de défit (*challenge output action*). L'utilisation de ce type d'actions permet de déterminer les secrets dynamiquement, dépendant de l'exécution du protocole. Ce cas est nécessaire par exemple en demandant à l'intrus de dériver une clef de session générée par un serveur et qui est nouvelle pour chaque session du protocole. Ce cas est présent pour l'exemple du protocole RA dans l'Exemple 3.4.1.1.

Exemple 3.4.1.1 *le protocole RA, modélisé dans [60].*
Pour le protocole RA [23] impliquant trois participants A, B et C, le message reçu par le serveur S est sous la forme $m = h_{K_c}(C, S, N_c, h_{K_b}(B, C, N_b, h_{K_a}(A, B, N_a, -))))$*, expliqué en Section 3.2. Le serveur doit manipuler récursivement cette liste de requêtes. Cette action récursive du serveur est simulée par un transducer, dont les transitions sont spécifiées comme suit :*

$$
\begin{array}{lcll}
start(*, h_{K_i}(P_i, S, x_0, x_1)) & \longrightarrow & read(v_N, h_{K_i}(P_i, S, x_0, x_1)) & (1)\\
read(v_R, h_{K_i}(P_i, P_j, x_0, -) & \longrightarrow & \{v_R, P_j, x_0\}_{K_i} & (2)\\
read(v_R, h_{K_i}(P_i, P_j, x_0, h_{K_l}(P_l, P_i, x_1, x_2))) & \longrightarrow & read(v_N, h_{K_l}(P_l, P_i, x_1, x_2)), & (3)\\
 & & \{v_R, P_j, x_0\}_{K_i}, & \\
 & & \{v_N, P_l, x_0\}_{K_i} &
\end{array}
$$

Dans cette spécification, x_0*,* x_1 *et* x_2 *sont des variables.* v_N *est une variable pour des constantes anonymes qui sert par exemple pour la règle* (1) *à modéliser la génération d'une clef de session qui sera stockée dans cette variable.* v_R *est une variable de registres qui sert à stocker une clef de session en passant d'une requête à une autre. Le transducer représentant l'action du serveur admet deux états : start et read. Dans l'état start, le transducer vérifie si la première requête est destinée à S et initialise le processus de lecture des requêtes en générant une clef de session qui sera stockée dans le registre. Dans l'état read, les requêtes sont traitées. Tout au long de cette phase, le registre (représenté par la variable* v_R*) est utilisé pour stocker une clef de session en passant d'une requête à une autre.*

Intrus et attaque.
L'intrus considéré est celui de Dolev Yao. Cependant, dans leur modèle, au contraire des participants, l'intrus ne peut pas générer de nouvelles constantes. Dans une attaque, l'intrus choisit un entrelacement entre les différentes actions *receive-send*, modélisées par des TTACs $(T_0, .., T_{l-1})$, tel que :

1. La dernière action T_{l-1} est une action de sortie de défit (*challenge output action*). Elle détermine le message secret m'_{l-1} que l'intrus doit dériver à partir de ses connaissances m_{l-1} acquises à la fin de l'action T_{l-1}. Ce message secret est une constante régulière ou anonyme. Il est présenté à l'intrus comme un défit mais qui n'est pas ajouté à ses connaissances (m'_{l-1} n'est pas inclus dans m_{l-1}).

2. L'intrus peut dériver l'entrée (m'_i) de chaque action T_i à partir des connaissances acquises des actions précédentes (m_i).

Résumé

[60] propose un algorithme de décision pour la vérification du secret pour les protocoles récursifs dans le modèle de Dolev Yao. La preuve de décidabilité du problème de secret est effectuée en deux étapes.

Dans la première étape, l'intrus est simulé par un TTAC appelé T_{der}. Dans une deuxième étape, supposons que les actions du protocole sont simulées par les TTACs T_0,...,T_{l-1}, les attaques sont décrites par des compositions successives de TTACs de la forme T_{der},T_{l-1},...,T_0,T_{der} (application de droite à gauche). T_{l-1} produit une paire (m, m'_{l-1}) où m représente l'ensemble des connaissances de l'intrus après cette étape. Quant à m'_{l-1}, il représente le défit que l'intrus essaye de dériver à partir de m sans utiliser m'_{l-1}. À la dernière étape, T_{der} essaye de produire une paire $\langle a, a\rangle$ avec a soit une constante soit une constante anonyme.

Le problème d'insécurité du protocole (existence d'une attaque) est réduit à un problème nommé *IteratedPreImage*. Ce problème consiste à décider, étant donné un terme t, un TTAC B et une séquence $T_0, .., T_{l-1}$, si t peut être dérivé à partir du langage reconnu par B en utilisant la séquence des TTACs $T_0, .., T_{l-1}$. Ce problème a été prouvé décidable au début du papier [60].

Limites.

Le modèle introduit dans [60] admet quelques limites :

- Les clefs considérées sont des clefs atomiques.
- Les règles sont linéaires à gauche. Ceci ne permet pas d'avoir de tests d'égalité. Or, plusieurs protocoles ne peuvent pas être modélisés dans ce contexte. D'ailleurs, même le protocole considéré comme exemple (RA), ne répond pas à cette contrainte.
- La relaxation de ces contraintes, i.e. le test d'égalité entre des messages arbitraires, ou l'ajout des opérateurs algébriques (exponentiation,..) ou le XOR, ou encore la manipulation des clefs composées, conduit à l'indécidabilité.

Les résultats d'indécidabilité dûs à la relaxation des contraintes ci-dessus sont prouvés par réduction du problème PCP (Post Correspondance Problem).

3.4.2 Clauses de Horn pour les protocoles récursifs (Truderung)

Truderung définit dans son modèle de [99] une classe bien définie de clauses de Horn, nommée théorie de sélection (*selecting theories*). Quand un message t est envoyé, il existe un symbole de prédicat r de telle manière que le traitement de ce message par un participant est manipulé par une théorie de sélection appliquée à $r(t)$.

Ce formalisme (de théorie de sélection) permet de modéliser les protocoles récursifs où les participants procèdent par des calculs récursifs ou itératifs. Il permet aussi à un participant d'envoyer un nombre non borné de messages.

Modèle

Nous nous intéressons dans ce paragraphe au modèle de protocoles considéré dans [99]. La signature Σ considérée comprend des constantes telles que des noms des participants, des nonces et des clefs, des symboles de fonctions unaires tels que la fonction de hachage et les symboles de fonctions binaires tels que les paires, les encryptions symétriques et asymétriques.

Soient Q et R deux ensembles disjoints de symboles de prédicats nommés respectivement *pop* et *push*. Une règle sur (Q, R) a la forme $t \to r(s)$. L'idée de base est que, après avoir reçu

un message $t\sigma$, pour une certaine substitution close σ, un participant répond par l'envoi des messages de $[[r(s\sigma)]]_\phi$.
$[[r(t)]]_\phi$ désigne les termes sélectionnés par ϕ, i.e. $[[r(t)]]_\phi = \{s \mid r(t) \vdash_\phi I(s)\}$, où ϕ est une théorie de sélection sur (Q, R). Une théorie de sélection est un ensemble de clauses de la forme :

$$\begin{array}{lll} q_1(x_1), \ldots, q_n(x_n) & \longrightarrow & q(f(x_1, \ldots, x_n)) \quad (1) \\ q_1(t), \ldots, q_l(t), r(t) & \longrightarrow & r'(x) \quad (2) \\ q_1(t), \ldots, q_l(t), r(t) & \longrightarrow & I(s) \quad (3) \end{array}$$

Dans la deuxième clause (2), x est une variable de t et t est flat, i.e., est de la forme $t = f(x_1, \ldots, x_n)$ où x_i pour $i = 1, .., n$ sont des variables. Dans la troisième clause (3), les variables de s sont incluses dans l'ensemble des variables de t. Ces trois types de clauses sont définis comme suit :

1. les clauses de type (1) sont nommées les clauses *pop*. Elles ont pour objectif de simuler les exécutions de n'importe quel automate non déterministe d'arbre fini ;
2. les clauses de type (2) sont nommées les clauses *push*. Elles transforment une information d'un terme à ses sous-termes. En effet, elles traitent les messages récursivement en allant d'un terme de ce message à un ou plusieurs sous-termes de ce terme.
3. les clauses de type (3) sont nommées les clauses *send*. Elles permettent à un participant d'envoyer des messages sur le réseau, et par la suite à l'intrus (vu que le modèle d'intrus retenu dans le modèle considéré est celui de Dolev Yao). $I(s)$ signifie que s est envoyé et donc il est connu par l'intrus.

Nous revenons à la définition de l'ensemble sélectionné par la théorie de sélection Φ, qui est défini comme suit : $[[r(t)]]_\Phi = \{s \mid r(t) \vdash_\Phi I(s)\}$ pour un terme t et $r \in R \cup I$. Nous avons $A \vdash_T B$ s'il existe une séquence de formules atomiques $a_1, \ldots, a_n$ tel que chaque élément de B est dans $a_1, \ldots, a_n$ et pour chaque i = 1,. . . ,n, on a deux cas :
- soit $a_i \in A$
- soit, il existe une clause $b_0 \leftarrow b_1, \ldots, b_m$ dans T et une substitution σ telles que
  - $a_i = b_0\sigma$
  - et chacune des $b_1\sigma, \ldots, b_m\sigma$ est dans $a_1, \ldots, a_{i-1}$.

Remarque. Les actions non récursives peuvent être traitées dans ce modèle puisque pour une théorie Φ, $[[I(s)]]_\Phi = \{s\}$. En effet, une règle de réécriture non récursive $t \longrightarrow s$ peut être écrite dans ce modèle sous la forme $t \longrightarrow I(s)$.

Participant et protocole. Un participant P est une séquence d'actions receive-send mais de la forme :
$P = (t_i \rightarrow r_i(s_i))_{i=1}^n$ tel que $t_i, s_i \in T(\Sigma, V)$ et $\forall v \in Var(s_i), v \in Var(t_1) \cup \ldots \cup Var(t_{i-1})$.

Un protocole sur (Q,R) est une paire (P,Φ) où P est un ensemble fini de participants et Φ est une théorie de sélection.

Exemple 3.4.2.1 *Le protocole RA, modélisé dans [99].*
Pour l'exemple de RA [23], les actions du serveur après avoir reçu toutes les requêtes des participants (sous la forme $m = h_{K_c}(C, S, N_c, h_{K_b}(B, C, N_b, h_{K_a}(A, B, N_a, -))))$, expliquée en Section 3.2, sont spécifiées comme suit :

$$\begin{array}{lcl} r(h_{K_i}(P_i, P_j, x, y)) & \longrightarrow & r(y) \\ r(h_{K_i}(P_i, P_j, x, h_{K_l}(P_l, P_i, x', y))) & \longrightarrow & I(\{K_{ij}, P_j, x\}_{K_i}), I(\{K_{il}, P_l, x\}_{K_i}) \\ r(h_{K_i}(P_i, P_j, x, -)) & \longrightarrow & I(\{K_{ij}, P_j, x\}_{K_i}) \end{array}$$

Il est à noter que, pour les actions du serveur, les actions récursives sont modélisées par des règles de la forme $x \longrightarrow r(x)$ *et donc la théorie de sélection n'utilise qu'un seul prédicat de push* r *i.e.* $\Phi = (\emptyset, \{r\})$.

Afin d'inclure les règles de l'intrus, la théorie Φ a été étendue en Φ_I qui représente la théorie du protocole (P,Φ). Concernant l'intrus, le modèle considéré est celui de Dolev Yao. Il y a donc les opérations de synthèse (composition des messages) et les opérations d'analyse (de décomposition de messages) habituelles. Elles sont définies dans le modèle [99] respectivement par le biais des clauses *send* et des clauses *pop*. Par exemple, l'analyse d'un message sous forme de paire $\langle x, y\rangle$ est assurée par les clauses *pop* suivantes : $r_I(\langle x, y\rangle) \longrightarrow r_I(x)$ et $r_I(\langle x, y\rangle) \longrightarrow r_I(y)$ tout en ayant une clause *send* qui permet d'envoyer tous les messages obtenus par décomposition d'un message. Cette clause a la forme suivante : $r_I(x) \longrightarrow I(x)$.

Quant aux opérations de synthèse, elles sont assurées par des clauses *send*. Par exemple, pour la même opération de paire, la clause est définie comme suit : $I(x), I(y) \longrightarrow I(\langle x, y\rangle)$.

Mis à part ces clauses, les clauses *push* et *pop* de Φ existent aussi dans la nouvelle théorie Φ_I. En outre, pour chaque clause *send* de Φ, une autre clause lui est associée dans la théorie Φ_I. À une clause de Φ de la forme $q_1(t), .., q_l(t), r(t) \longrightarrow I(s)$, correspond une des deux clauses suivantes :

- *règle généralisée push* : $I(k_1), ..., I(k_n), q_1(t), q_l(t), r(t) \longrightarrow r_I(x)$
- *règle généralisée send* : $I(k_1), ..., I(k_n), q_1(t), q_l(t), r(t) \longrightarrow I(s')$

avec s' (respectivement x qui est une variable) est un terme accessible de s tout en traversant les clefs k_1,..,k_n qui sont connues dans les hypothèses de ces deux clauses.

Exécution, attaque

Une exécution de protocole est une séquence de règles : $\pi = \pi_1, ..., \pi_n$. Chaque élément de π est associé à un participant $P_1, ..., P_l$.

Pour un participant P_k, la sous séquence des éléments de π assignée à P_k est $P_k^1, ..., P_k^m$ où $m < |P_k|$. Une attaque est représentée par la paire (π, σ) où $\pi = (ti \longrightarrow r_i(s_i))_{i=1}^n$ est une exécution du protocole et σ est une substitution close. L'idée est toujours la même : l'intrus peut construire t_i à partir des t précédents (modulo une substitution close). Quand l'exécution termine, il doit récupérer le secret($I(Sec)$).

Néanmoins, dans ce travail, seule la propriété de secret est traitée. Cependant, pour les protocoles de groupe, il y a des propriétés de sécurité très importantes à vérifier. Par exemple, pour les protocoles d'accord de clefs, une propriété assez importante à vérifier est celle d'accord (*agreement*) de clef. En effet, l'intrus n'a pas besoin d'avoir la clef du groupe pour attaquer le protocole. Il suffit juste qu'il fasse en sorte qu'au moins deux participants n'aient pas la même vue de clef du groupe.

Avantages

[99] introduit un nouveau formalisme pour modéliser les protocoles cryptographiques récursifs. L'idée principale derrière ce formalisme est que, chaque message est le résultat d'application de

la réécriture dans certains sous-termes. Pour déterminer quelle règle de réécriture et quel sous-terme, il faut utiliser une théorie de sélection. Au contraire de systèmes d'automates d'arbres, le modèle proposé peut exprimer les protocoles dont la partie gauche des règles n'est pas forcément linéaire. Ainsi, ce modèle offre la possibilité de stocker et comparer les messages en utilisant un système de réécriture. Dans ce modèle, les participants peuvent donc recevoir des messages de taille non bornée, envoyer des messages multiples et comparer et stocker des messages. Ce modèle utilise une classe de clauses de Horn pour modéliser les calculs récursifs des participants.

Le modèle introduit dans [99] présente une procédure de décision pour vérifier si un protocole satisfait la propriété de secret. L'algorithme dans ce modèle est NEXPTIME et est basé sur une dérivation d'une borne exponentielle de la taille des attaques minimales. Ce résultat est prouvé par l'existence d'une attaque minimale de taille exponentielle. Un ADAG (*direct acyclic graph of the attack*) est associé à une attaque d'un protocole, si elle existe. En montrant que s'il existe un ADAG représentant une attaque alors il existe un ADAG minimal de taille exponentielle, ceci a pour conséquence d'avoir un algorithme NEXPTIME pour décider l'insécurité des protocoles.

Limites

Les clefs utilisées dans les encryptions à clefs asymétriques ou à clefs symétriques sont des constantes (clefs atomiques). En outre, le modèle de Truderung permet la manipulation d'un nombre fini de constantes. Il n'y a pas de mécanisme permettant à un participant de générer un nombre non borné de nonces ou de clefs de sessions par exemple. Cette restriction empêche la modélisation de protocoles tels que le protocole RA. Dans ce protocole, le serveur doit pouvoir traiter un message de taille arbitraire vu qu'il n'a pas connaissance de la taille du message qu'il va recevoir. Le serveur doit générer des clefs fraîches pour chaque requête. Cependant, dans le modèle [99], la modélisation des actions du serveur du protocole RA définissent une clef constante pour chacune des paires des participants. Cependant, ceci mène à la réutilisation des mêmes clefs, ce qui est loin d'être la réalité. Par exemple, en recevant le message

$$h_{K_B}(B, S, N'_B, h_{K_A}(A, B, N'_A, h_{K_B}(B, A, N_B, h_{K_B}(A, B, N_A, -)))),$$

le serveur doit générer une clef pour chaque requête . Ceci est indépendant du fait que certaines requêtes sont destinées aux mêmes participants. Par exemple, pour les participants A et B, le serveur doit générer deux clefs partagées K_{AB} et K'_{AB} pour les deux requêtes. Néanmoins, dans le modèle [99], le serveur a une seule clef partagée K_{AB} pour les deux requêtes.

Ensuite, dans le modèle [99], on ne peut pas modéliser des calculs tels que le mappage de listes (*list mapping*) ou celui de symboles fonctionnels (*functional symbols mapping*). Le premier mapping sert à produire une liste encodée $\{[t'_1, \ldots, t'_n]\}_{k'}$ à partir d'une liste encodée $\{[t_1, \ldots, t_n]\}_k$ où, pour chaque $i = 1, \ldots, n$, le terme t'_i est le résultat d'application d'une règle de réécriture simple à un terme t_i. Le second mappage remplace les symboles fonctionnels d'un certain terme par des symboles fonctionnels de même arité, tout en conservant la même structure du terme, i.e. les occurrences distinctes d'un même symbole doivent être remplacées par un même symbole.

Autrement, le modèle peut modéliser le mappage d'une liste $[t_1, \ldots, t_n]$ à une autre liste $[t'_1, \ldots, t'_n]$ où t'_i est le résultat d'application de certaines règles de réécriture au terme t_i. Ceci est dû au fait que, du point de vue du modèle de Dolev Yao, le fait d'envoyer $[t'_1, \ldots, t'_n]$ a le même effet qu'envoyer les termes $t'_1, \ldots, t'_n$ séparément.
Cependant, le résultat présenté en [99] est un résultat théorique et la possibilité d'implémenter cette procédure reste une question ouverte.

3.4.3 Extensions du modèle de Truderung (Küsters-Truderung, Kürtz)

Le travail de Truderung [99] a présenté un résultat de décidabilité pour l'insécurité des protocoles récursifs avec théories de sélection en considérant un nombre borné de sessions. Ce travail a été étendu pour tenir compte d'autres propriétés des protocoles récursifs. Nous présentons dans le premier paragraphe l'extension faite par Truderung et Küsters [59] pour tenir compte de l'opérateur XOR omni-présent dans la plupart des protocoles récursifs. Dans le deuxième paragraphe, Kurtz [58] modélise des constantes fraîches (et par la suite des clefs fraîches), utilisées par exemple dans le cas du protocole RA [23].

Première extension du modèle de Truderung

Küsters et Truderung [59] étendent le modèle de Truderung [99] en ajoutant l'opérateur ou exclusif (XOR ou $\oplus$). L'intérêt de cet opérateur est motivé par la caractérisation des protocoles de groupe d'avoir recours à cet opérateur pour dériver des clefs à base des opérations algébriques comme c'est le cas des protocoles de Diffie-Hellman de groupe (GDH, A-GDH, ...).

Modèle. La signature est la même que celle de [99] avec en plus l'opérateur XOR ($\oplus$). Ils considèrent donc une théorie équationnelle représentant ce nouvel opérateur :

$$\begin{array}{lcl} x \oplus y & = & y \oplus x \\ (x \oplus y) \oplus z & = & x \oplus (y \oplus z) \\ x \oplus x & = & 0 \\ x \oplus 0 & = & x \end{array}$$

Protocole, étape de protocole : Un protocole P est un tuple $(\Pi_1, \ldots, \Pi_l)$ de participants Π_i. Un participant Π est un arbre fini où chaque arête est étiquetée par une étape du protocole. Une étape du protocole consiste en une règle de protocole et un programme d'envoi (send program). Cette étape exprime que, quand le participant reçoit un message, il va envoyer un ensemble de messages en exécutant un programme d'envoi correspondant à cette étape. Une règle de protocole est de la forme $t \rightarrow q(s)$ où q est un symbole de prédicat unaire. Un programme d'envoi ϕ est une théorie de Horn unaire où chaque clause est de la forme :

$$\begin{array}{lcll} q_1(t) & \Rightarrow & q_2(x) & (1) \\ q_3(s) & \Rightarrow & I(s') & (2) \end{array}$$

Dans la première clause (1), x est une variable de t, t est linéaire (chaque variable de t se produit au plus une seule fois dans t) et ne contient pas le symbole $\oplus$. Dans la deuxième clause (2), les variables de s' sont incluses dans s. q_1, q_2 et q_3 sont des symboles de prédicats unaires.

Dans ce modèle, les ensembles des variables des règles du protocole des différents participants sont disjoints. Les ensembles des prédicats des différents programmes d'envoi sont différents à l'exception du prédicat I. Les déductions de l'intrus sont étendues par une règle concernant l'opérateur $\oplus$ permettant à l'intrus de composer un terme $x \oplus y$ s'il a connaissance de x et de y.

Résultat. Les auteurs de [59] montrent en premier lieu que l'ajout naïf d'un opérateur XOR au modèle de Truderung conduit nécessairement à l'indécidabilité du problème d'insécurité. Ceci est prouvé en codant le problème de PCP (Post Correspondance Problem) en considérant un seul participant avec une seule étape du protocole. Plus précisément, le résultat d'indécidabilité se base sur le fait que les participants du protocole unissent en XOR des messages arbitraires reçus du réseau. Les auteurs ont obtenu aussi un résultat d'indécidabilité dans le cas de l'utilisation

des clefs composées. Ils montrent ensuite la décidabilité du problème d'insécurité dans le cas des protocoles dits ⊕linéaires. Il s'agit des protocoles où les participants composent des messages en XOR mais en utilisant seulement un message fixe, i.e. ne provenant pas des messages reçus du réseau et un autre message provenant des messages reçus du réseau. D'une manière formelle, un protocole P est dit ⊕linéaire si dans chaque sous-terme de la forme $t \oplus s$ dans P, t ou s est clos. Si par exemple P comporte le terme $(x \oplus a) \oplus y$ avec a clos et x et y des variables alors P n'est pas ⊕linéaire.

La preuve de décidabilité est faite par réduction du problème d'insécurité dans le modèle de théorie de Horn avec XOR en un autre modèle sans présence du XOR, qui est déjà prouvé décidable d'après [99]. Pour effectuer cette transformation, certaines propriétés doivent être satisfaites par les dérivations dans le modèle de théorie de Horn avec le XOR. Dans cette transformation, la capacité des participants à faire des calculs récursifs/itératifs sert à imiter les applications de l'opérateur XOR.

Notons que, comme le modèle de Truderung [99], le nouveau modèle n'utilise que des clefs atomiques. Ils considèrent aussi un nombre borné de sessions.

Deuxième extension du modèle de Truderung

[58] étend le modèle de Truderung [99] pour pouvoir générer un nombre non borné de nonces et de clefs fraîches. Cette caractéristique est intéressante surtout pour les protocoles récursifs tels que le protocole RA. D'ailleurs, la modélisation de ce protocole dans le nouveau modèle introduit dans [58] permet de résoudre le premier problème détaillé en Section 3.4.3.

Modèle Cette extension consiste en premier lieu à étendre la signature par addition d'un ensemble non borné (Γ), nommées constantes anonymes, à l'ensemble des constantes Σ. L'ensemble de variables X est aussi étendu en ajoutant deux ensembles de variables : les variables anonymes (Y) et les variables fraîches (Y^*). Les termes sont construits sur tous ces ensembles de constantes et de variables ($T_V = (\Sigma \cup \Gamma, X \cup Y \cup Y^*)$).

Le modèle utilise la notion de registre qui sert à stocker les constantes anonymes. Il représente donc une mémoire non bornée. L'objectif est que, ces participants soient capables de mémoriser les constantes anonymes d'une étape itérative à une autre. Ainsi, en traitant un message, un participant peut accéder à une mémoire spéciale appelée séquence de registres, qui consistera en un nombre fini de registres. Ces registres comportent des constantes anonymes. Les participants peuvent lire des constantes de cette mémoire, en éliminer d'autres et générer des constantes anonymes dans n'importe quel registre.

Quant aux clauses de Horn utilisées dans ce modèle, d'une manière similaire au modèle de Truderung, il existe les mêmes trois types de clauses de Horn : les clauses pop, push et send. Cependant, au contraire du modèle de Truderung qui utilise des symboles de prédicats seulement unaires, dans le nouveau modèle, il existe des symboles de prédicats unaires (Q et I) et d'autres binaires (R). Ce dernier type de prédicat va contenir le registre. Il existe une deuxième différence par rapport au modèle de Truderung. En effet, les termes qui existent dans la partie gauche des clauses de Horn sont soit linéaires soit plat (flat). Un terme t est linéaire si chaque variable de t se produit au plus une fois dans t. Un terme t est plat si t est de la forme $f(x_1, .., x_n)$ où x_i, $i = 1..n$ sont des variables non nécessairement distinctes.

Exemple 3.4.3.1 *Le protocole RA, modélisé dans [58].*
Nous revenons à l'exemple de RA [23] pour illustrer cette extension. L'action du serveur est représentée par une seule étape du protocoles $x \longrightarrow r(x)$ *tout en ayant comme théorie de sélection* $(\emptyset, \{r\})$ *utilisant des constantes anonymes avec deux registres. Cette théorie de sélection est alors*

donnée par les clauses suivantes :

$$\begin{array}{lcl} r(h_{K_i}(P_i, P_j, x_1, x_2), (y_1, y_2)) & \longrightarrow & r(x_2, (y_2, y^*)) \\ r(h_{K_i}(P_i, P_j, x_1, h_{K_l}(P_l, P_i, x_2, x_3)), (y_1, y_2)) & \longrightarrow & I(\{y_1, P_j, x_1\}_{K_i}), I(\{y_2, P_l, x_1\}_{K_i}) \\ r(h_{K_i}(P_i, P_j, x_1, -), (y_1, y_2)) & \longrightarrow & I(\{y_1, P_j, x_1\}_{K_i}) \end{array}$$

Dans cette spécification, la première clause passe du traitement d'une requête à une autre. D'où la génération d'une nouvelle constante anonyme qui sera stockée dans y^*. Dans la deuxième clause, le serveur génère deux certificats dont les clefs correspondent aux constantes anonymes stockées dans les deux registres. La dernière clause traite la dernière requête en générant un certificat correspondant à la constante anonyme stockée dans le premier registre.

Résultat [58] propose un algorithme de décision d'insécurité des protocoles utilisant les constantes anonymes. Cette décision est non déterministe en un temps double exponentiel. Le résultat de ce travail est obtenu pour une taille non bornée de messages et un nombre non borné de clefs et nonces frais. Il est cependant valide pour un nombre borné de sessions. En outre, le nombre de participants est limité permettant seulement une seule session par participant. La preuve de ce résultat est faite en deux étapes. La première étape consiste à construire un ADAG qui représentera une éventuelle attaque du protocole. La transformation de la définition d'une attaque consiste en deux pas. Dans le premier pas, une théorie de Horn appelée théorie du protocole est définie en combinant la théorie de sélection du protocole et la théorie de l'intrus. Cette théorie de protocole permet d'obtenir directement les ajouts d'informations aux connaissances de l'intrus à partir d'une étape du protocole $t \longrightarrow r(s)$. Le deuxième pas de la transformation consiste à étendre la théorie du protocole en une théorie d'étapes (stage theory) qui permet de modéliser la totalité des exécutions du protocole dans une seule application de la théorie.

Dans une seconde étape, le travail montre que, pour chaque ADAG, il existe un autre ADAG d'une taille bornée en double exponentiel, résultant de la possibilité de deviner et d'examiner chaque ADAG de manière non déterministe et en un temps double exponentiel.

3.4.4 Décidabilité dans le cas d'un intrus passif (Kremer et al.)

Récemment, [57] ont développé une technique d'approximation à base d'automates qui permet d'analyser la classe des protocoles de groupe et de vérifier l'absence d'éventuelles attaques en présence d'un intrus **passif**. Cet intrus ne fait qu'écouter tous les messages émis durant toute session.

Modèle

Mis à part les constantes, la signature Σ comprend des symboles de fonctions binaires telles que la paire (*pair*), l'encryption (*enc*), l'exponentielle (*exp*), la multiplication (*mult*) et le ou exclusif (*xor*). Elle comprend aussi un symbole de fonction unaire qui représente la hachage (H).

Ce modèle est étendu par une théorie équationnelle représentée par un système de réécriture R modulo associativité et commutativité.

$$\begin{array}{lcl} x \oplus 0 & \rightarrow & x \\ x \oplus x & \rightarrow & 0 \\ ((x)^y)^z & \rightarrow & x^{y.z} \\ \langle x, y \rangle^z & \rightarrow & \langle x^z, y^z \rangle \end{array}$$

Protocole. Un protocole est défini par deux fonctions e et k. Pour un protocole à n participants, ce protocole est défini par $e(n)$ et $k(n)$. $e(n)$ désigne l'ensemble des termes émis par les participants du protocole durant une exécution de ce protocole. $k(n)$ désigne l'ensemble de termes qui doivent être secrets durant cette exécution. Ces ensembles de termes peuvent être définis de manière inductive. Un exemple de l'ensemble $k(n)$ serait de considérer l'ensemble à un singleton qui représente la clef à construire. Une spécification du protocole est une paire d'ensembles de termes (E, K) telle que $E = \bigcup_{n\in\mathbb{N}} e(n)$ et $K = \bigcup_{n\in\mathbb{N}} k(n)$.

Intrus et propriétés. L'intrus considéré dans ce modèle est l'union des déductions de l'intrus de Dolev Yao (DY). Le système complet de déductions est nommé I. Mis a part les règles standard de DY (analyse et synthèse), I est étendu par deux autres règles additionnelles. La première règle permet à l'intrus de composer un terme ${t_1}^{t_2}$ avec t_2 est une constante. L'intrus peut composer ce message s'il peut composer les deux termes t_1 et t_2. La deuxième règle permet à l'intrus de composer un terme $t_1 \oplus ... \oplus t_n$ si l'intrus arrive à composer chacun des termes t_i. Après chaque application d'une règle, le résultat est mis en forme normale modulo associativité et commutativité.

Le modèle considéré focalise sur la propriété de secret d'un certain ensemble de messages. Dans le cadre des protocoles d'échange de clefs de groupe, l'ensemble de messages qui doivent être secrets serait l'ensemble des clefs de sessions établies durant les différentes sessions du protocole. Dans ce modèle, cette propriété peut être définie comme suit : étant donné une spécification du protocole (E, K), existe-t-il un secret de K qui peut être déduit d'un certain ensemble émis dans une session et donc de E ? La vérification de cette propriété revient donc à tester si $I(E) \cap K = \emptyset$.

Résultats

Les auteurs introduisent des restrictions pour les spécifications des protocoles permettant de réduire les capacités de l'intrus. Les déductions de l'intrus sont donc réduites de I à celles de DY pour une classe particulière de spécifications de protocoles appelées spécifications *bien formées*. Une spécification de protocole (E, K) est dite bien formée si elle satisfait trois conditions. La première condition précise que l'opérateur $\oplus$ a une arité 2 et se produit uniquement en position racine de tout terme de E. En outre, les composants u et v d'un terme $t = u \oplus v$ de E ne doivent pas être déductibles par l'intrus. La deuxième condition précise que toute constante figurant comme exposant dans un terme de $E \cup K$ ne doit pas être accessible et donc ne doit pas être dans la sur-approximation des termes accessibles. La troisième condition nécessite que les secrets ne contiennent pas des termes construits à base de $\oplus$, i.e. l'ensemble de clefs n'impliquent pas de $\oplus$. Le papier montre que, pour une classe de protocoles bien formés, un modèle des capacités de l'intrus impliquant l'exponentiation et le $\oplus$ est équivalent à un modèle plus faible qui peut être vu comme le modèle de DY modulo associativité et commutativité de certains opérateurs.

Par des séries de réductions et de sur-approximations, le problème d'insécurité d'un protocole de groupe en présence d'un intrus passif est prouvé en utilisant des techniques d'automates d'arbre avancés. Pour représenter une sur-approximation de l'ensemble de messages émis et l'ensemble des secrets, les auteurs utilisent un formalisme d'automates d'arbre avancé appelé VTAM pour *Visibly Tree Automata with Memory, visibly structural constraints*. L'ensemble de messages que l'intrus peut déduire d'un ensemble de messages émis peut être décrit de nouveau dans ce formalisme. En outre, montrer que ces deux ensembles sont disjoints est décidable dans ce formalisme. Ce formalisme permet aussi de spécifier les ensembles définis inductivement de

messages des protocoles de groupe.

La signature Σ utilisée dans la spécification de protocole peut être infinie vu qu'elle contient les constantes indicées selon le nombre de participants dans une session. Pour définir un automate reconnaissant E et K, les auteurs introduisent une signature Σ' qui va contenir uniquement des constantes, des symboles binaires et une fonction appropriée ρ qui associe aux termes de Σ des termes de Σ'. Vu que Σ' est finie, les auteurs peuvent alors tester si $\rho(DY(E)) \cap \rho(K) = \emptyset$ en utilisant les automates avancés.

Pour représenter l'associativité et la commutativité de certains opérateurs dans leur modèle d'automates, les auteurs introduisent une fonction nommée W qui permet d'associer à tout terme t' de Σ' de la forme $t' = \rho(t)$, le plus petit élément de ρ de sa classe d'équivalence modulo AC, ceci pour un certain ordre qu'ils ont défini sur Σ'.

Finalement, les auteurs ont montré que le langage reconnu par la classe des automates considérée est préservé par construction de la clôture de DY.

3.4.5 Discussion

Nous résumons dans la Table 3.4.5 les différents travaux théoriques qui ont considéré les protocoles récursifs. Pour chacun de ces travaux, nous rappelons le modèle qu'il utilise ainsi que les limites de ce modèle.

Référence	**Modèle**	**Limites**
Küsters [60]	automates d'arbres.	clefs atomiques. règles linéaires à gauche.
Truderung [99]	théorie de sélection : une classe de théorie de Horn.	nombre de sessions borné. clefs atomiques. pas de manipulation de mappage de liste.
Küsters-Truderung [59]	extension du modèle de Truderung en ajoutant l'opérateur XOR.	nombre de sessions borné. clefs atomiques.
Kürtz [58]	extension du modèle de Truderung en permettant la fraîcheur des clefs et des nonces.	clefs atomiques.
Kremer et al. [57]	automates d'arbre avancés.	intrus passif.

Tab. 3.1 – Résumé des résultats de décidabilité pour les protocoles récursifs

3.5 Notre expérimentation préliminaire

Nous nous intéressons dans cette section à la recherche d'attaques sur des protocoles de groupe. Comme nous l'avons décrit en Section 3.3, plusieurs travaux [81, 73, 94, 96] ont été menés dans ce sens. La plupart de ces travaux ont mené à la découverte de quelques attaques pour certains protocoles. Bien que ces analyses ont été destinées à la base à vérifier les protocoles pour n'importe quel nombre de participants, la plupart des attaques ou des contre-exemples

donnés [81, 94] sont encore valides pour un petit nombre de participants (un, deux, trois et quatre participants).

Cette constatation a motivée notre première expérimentation préliminaire qui a consisté à vérifier quelques protocoles du corpus [3] à l'aide d'un outil automatique de vérification de protocoles cryptographiques. Nous avons commencé par étudier quelques protocoles de groupe, plus particulièrement quelques protocoles d'accord de clefs et de distribution de clefs dans le but de retrouver des attaques connues pour ces protocoles ou bien de trouver de nouvelles attaques. Nous avons donc considéré différents scénarii de quelques protocoles tels que les protocoles Asokan-Ginzboorg, GDH, Tanaka-Sato ou Iolus.

Nous détaillons en Sections 3.5.2, 3.5.3, 3.5.4, 3.5.5 les différents protocoles analysés. Pour chacun de ces protocoles, nous décrivons informellement les différents rôles mis en jeu et les actions qu'ils doivent effectuer. Nous décrivons ensuite les scénarii considérés, les problèmes rencontrés ainsi que les résultats trouvés. Le détail des spécifications en HLPSL est donné en Chapitre A en Annexe. Nous avons également pu spécifier et vérifier une architecture de protocoles dits hiérarchiques, i.e. dont la structure est modélisée sous forme de classes et où la communication est étroitement liée à cette structuration du groupe. Une attaque de milieu a été trouvée pour cette modélisation et une correction a été donc proposée. Cette analyse est résumée en Section 3.5.6. Finalement, un bilan de tous les tests effectués sera donné en Section 3.5.7.

3.5.1 Méthode

Notre objectif est de considérer des scénarii bien particuliers de quelques protocoles de groupe. Pour vérifier ces différents protocoles, nous avons opté pour l'outil AVISPA [4] (*Automated Validation of Internet Security Protocols and Applications*) ayant quatre back-ends illustré en Figure 3.6. En particulier, nous avons utilisé les deux back-ends CL-AtSe [102] et OFMC [10].

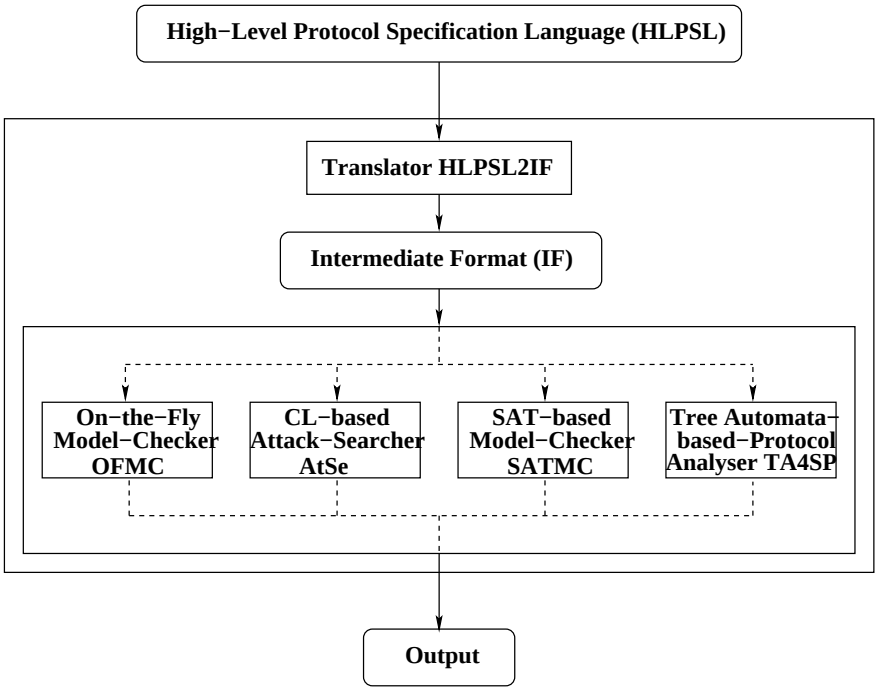


FIG. 3.6 – Architecture de l'outil AVISPA

Il est à noter que, pour quelques scénarii des protocoles, nous avons essayé d'utiliser les autres back-ends de AVISPA mais ces derniers n'arrivent pas à décider si les protocoles considérés sont corrects ou non. Nous nous sommes donc limités aux back-ends CL-AtSe et OFMC.

L'entrée de l'outil AVISPA, et donc de ses back-ends, est une spécification du problème d'insécurité considéré, i.e. le protocole à étudier ainsi que la propriété à vérifier, en langage HLPSL [25]. Ce langage est basé sur les rôles. Nous distinguons deux types de rôles : les rôles de bases qui représentent les participants et les rôles composés qui représentent des scénarii impliquant ces différents rôles de base. Les rôles de base détiennent quelques informations passées en paramètres et communiquent avec les autres via des canaux de Dolev Yao. Les actions d'un rôle de base sont modélisées comme un ensemble des transitions décrivant les changements d'états de ce rôle selon des évènements et des faits. Ces rôles de base sont combinés en parallèle ou en séquence pour former les rôles composés tels que les rôles de session ou d'environnement. Le rôle d'environnement sert à passer les paramètres aux autres rôles. Mis à part ces rôles de base et combinés, la spécification HLPSL doit comporter aussi une partie dédiée à la spécification des propriétés à vérifier. Les propriétés considérées sont le secret et l'authentification ainsi que d'autres propriétés d'atteignabilité.

La spécification en HLPSL est traduite automatiquement en une spécification en langage intermédiaire (IF) par le biais du traducteur HLPSL2IF. Cette spécification est donnée par la suite aux back-ends pour chercher d'éventuelles attaques pour le scénario considéré.

3.5.2 Analyse du protocole Asokan-Ginzboorg

Nous nous intéressons dans cette section à la modélisation du protocole de Asokan-Ginzboorg [6]. Nous présentons tout d'abord les problèmes rencontrés pour modéliser ce protocole. Ensuite, nous spécifions en HLPSL les scénarii considérés.

Problèmes liés à la modélisation du protocole Asokan-Ginzboorg

Le protocole Asokan-Ginzboorg [6] décrit la génération d'une clef de session entre un leader du groupe et un nombre arbitraire de participants. Nous disposons donc de deux rôles principaux : un leader et un membre du groupe.

Lorsque le leader déclenche l'exécution du protocole en envoyant la clef d'encryption (e), chaque participant génère deux informations (une clef symétrique r_i et une contribution à la clef du groupe s_i) et les envoie encryptées par la clef reçue (e). De même, quand un membre reçoit le dernier message du serveur et calcule sa clef, il renvoie une confirmation de cette clef. Ainsi, du point de vue d'un participant m_i du groupe, les actions sont bien définies sans ambiguité. Elles se résument en deux actions *receive-send*, définies somme suit :

$$l, \{e\}_p \longrightarrow \{r', s'\}_e$$
$$\{x\}_r \longrightarrow m_i, \{s', h(x)\}_{f(x)}$$

La première action sert à envoyer la contribution et la clef du participant considéré (r', s') lorsqu'il reçoit la clef e du serveur, encryptée par la clef p, connue par tous les membres du groupe. Dans la deuxième action, le participant envoie sa confirmation de la clef du groupe dès qu'il reçoit du serveur le message x qui modélise la concaténation des contributions de tous les membres du groupe.

Concernant le rôle du serveur, après avoir envoyé la clef d'encryption, il doit récupérer les différentes contributions de tous les participants. Pour cela, il a besoin d'effectuer un tra-

vail itératif : il est en attente des contributions de **tous** les membres du groupe. Deux cas se présentent :

1. Le serveur connaît la composition du groupe (le nombre de membres et leur identité). Concernant l'étape de la collecte des contributions, le serveur profite de sa vue du groupe pour tester le point de déclenchement de la prochaine étape (génération de la clef du groupe à partir des contributions). Ce point est atteint quand le serveur a reçu les contributions de l'ensemble des membres. Pour ce cas, on a besoin d'instancier le groupe dès le départ afin de donner au serveur la liste des membres du groupe.
2. Le serveur ne sait pas la composition du groupe. Il attend les contributions des participants. À chaque réception d'une contribution d'un agent, le serveur ajoute cet agent au groupe (génération d'un nouveau rôle de membre de groupe). Dans ce cas, la question qui se pose est quand le serveur arrête-t-il d'attendre les contributions des agents pour pouvoir passer à l'étape de génération de la clef ? Une solution à ce problème serait d'envisager qu'à chaque fois qu'il reçoit une contribution, le serveur calcule sa nouvelle clef de session et la transmet au groupe. Néanmoins, cette variante du protocole ne suit pas la description informelle du protocole.

Dans le reste de cette section, nous gardons donc le premier cas : le serveur connaît l'ensemble des agents appartenant au groupe. Ainsi, le problème concernant la deuxième étape du protocole est la modélisation de l'action itérative du serveur (recevoir **toutes** les contributions des membres du groupe).

La troisième étape des actions du serveur est la génération de la clef du groupe. Cette clef est obtenue par concaténation des contributions des différents membres du groupe ainsi que celle du serveur. Deux cas se présentent :

1. Puisque cette étape se fait après avoir terminé la collecte des contributions, une manière de faire est de conserver les contributions puis de les utiliser ensuite pour la génération de la clef.
2. Nous pouvons aussi envisager que cette étape se fait en parallèle avec la deuxième étape. En effet, en recevant une contribution d'un agent, le serveur peut mettre à jour la clef de session en y concaténant la nouvelle contribution.

Nous optons pour le deuxième cas puisqu'il est plus simple à faire et demande moins de calculs.

La quatrième étape consiste à envoyer la clef de session à chaque membre du groupe encrypté par la clef symétrique correspondante. Elle nécessite la sauvegarde des clefs symétriques collectées dans la phase de contribution. Cette étape demande aussi un travail itératif : envoyer à **tous** les membres du groupe la clef générée. Le passage à l'état 5 est effectué quand tous les messages destinés à tous les membres du groupe sont envoyés par le serveur. Cette étape consiste à la récupération des messages encryptés par la nouvelle clef du groupe de **tous** les membres du groupe.

Notons que la plupart des étapes (2, 4 et 5) du serveur repose sur un traitement itératif : soit pour récuperer les contributions de **tous** les membres du groupe, soit pour envoyer un message destiné à **chacun** des participants du groupe. Pour modéliser ce protocole, il faut donc manipuler des structures de données permettant de sauvegarder et de traiter certaines informations concernant la structure du groupe.

Le protocole Asokan-Ginzboorg en général

Nous considérons dans ce paragraphe, le protocole Asokan-Ginzboorg en supposant que le nombre de participants impliqués est donné en paramètre dans le rôle composé *Environnement*. Nous modélisons donc le protocole par deux rôles de base : un rôle de leader et un rôle de membre ordinaire. Le rôle du membre ordinaire peut être joué plusieurs fois selon le paramètre spécifié dans le rôle composé. La gestion du groupe ainsi que le calcul de la clef a été fait d'une manière itérative. Pour cela, nous avons opté pour l'utilisation des ensembles. Ces ensembles ont été utilisés pour le passage au rôle serveur de quelques paramètres liés à la composition du groupe tels que les membres du groupe. Ils sont aussi utilisés comme structure de données intermédiaires afin de manipuler quelques informations et vérifier des conditions telles que la condition de passage de la phase de collecte de contribution à la phase de génération de la clef. Par exemple, la phase de récolte des contributions est spécifiée par trois étapes. La première étape consiste à initialiser la phase de collecte des contributions. Elle se déclenche à la réception de la première contribution d'un certain participant. Cette étape est spécifiée en HLPSL de la manière suivante :

```
 step2. State=1 /\ Rcv(M'.{R'.S'}_E)
/\ in(M',Membersbis)
   =|> State':=2
       /\ MemberKeys':=cons(M'.R',MemberKeys)
       /\ MemberContribs':=cons(M'.S',MemberContribs)
       /\ Sall':=S'
       /\ Membersbis' := delete(M',Membersbis)
```

Les ensembles $MemberKeys$ et $MemberContribs$ servent à sauvegarder respectivement l'ensemble des clefs et l'ensemble des contributions des membres du groupe. La variable *Sall* sert à sauvegarder la concaténation des contributions des membres du groupe. C'est la valeur de cette variable, concaténée avec la contribution du serveur qui va être envoyée ensuite aux différents membres pour qu'ils calculent la clef du groupe. L'ensemble $Membersbis$ est un ensemble intermédiaire qui sert à détecter la fin de la phase de récolte des contributions. À la réception d'un message contenant la contribution et la clef d'un participant du groupe, le serveur va mettre à jour les ensembles de clefs et de contributions en ajoutant respectivement à ces deux ensembles la clef et la contribution reçues. L'identité du participant ayant envoyé sa contribution est retirée de l'ensemble $Membersbis$ afin d'utiliser cet ensemble ensuite pour vérifier la condition d'arrêt de cette phase.

Une fois que la phase de récolte est déclenchée, le serveur continue de recevoir les contributions des différents membres tant qu'il n'a pas eu toutes les contributions attendues. Cette condition est garantie par un test vérifiant que le membre envoyant la contribution existe encore dans l'ensemble $Membersbis$ (i.e. ce membre n'a pas encore contribué). Cette deuxième étape est donc spécifiée par cette boucle :

```
step3. State=2 /\ Rcv(M'.{R'.S'}_E)
               /\ in(M',Membersbis)
  =|> State':=2
      /\ MemberKeys':=cons(M'.R',MemberKeys)
      /\ MemberContribs':=cons(M'.S',MemberContribs)
      /\ Sall':= Sall.S'
      /\ Membersbis' := delete(M',Membersbis)
```

Cette spécification exprime que, en recevant un message de contribution d'un membre qui n'a pas encore contribué (il appartient à $Membersbis$), la clef de ce membre est ajouté à l'ensemble de clefs ($MemberKeys$), la contribution est ajouté à l'ensemble de contributions ($MemberContribs$), et la valeur de variable ($Sall$) prévue pour la concaténation des contributions de tous les membres du groupe est mise à jour.

La phase de récolte des contributions termine lorsque toutes les membres ont contribué et donc l'ensemble $Membersbis$ ne contient plus d'éléments. Cette étape est modélisée en HLPSL comme suit :

```
step4. State=2 /\ not(in(M2,Membersbis))
  =|> State':=3
      /\ SL' := new()
      /\ Sall':= Sall.SL'
```

La variable $Sall$ contient maintenant les contributions de tous les membres du groupe, y compris celle du leader (qui est fraîchement générée). À ce stade, une autre phase itérative peut commencer, celle de l'envoi de la valeur de $Sall$ à tous les membres du groupe en utilisant l'ensemble des clefs construit dans la phase de récolte de contributions : $MemberKeys$. La spécification de tout le protocole est décrite en Section A.1.1 en Annexe.

Vu que le but de ce protocole est la génération d'une clef de groupe commune entre le serveur et les participants, la propriété considérée dans la spécification adoptée est le secret de cette clef de groupe. Le résultat de la vérification de la spécification, considérée dans ce paragraphe, par AtSe ne donne aucune attaque. Nous avons pourtant vérifié que la spécification de l'entrée de cet outil correspond à notre spécification en HLPSL.

Instance du protocole Asokan-Ginzboorg

Nous présentons dans ce paragraphe une instances du protocole étudié. Notre objectif était dans un premier temps de retrouver les attaques [94] qui ont été trouvées pour certains scénarii. L'attaque que nous visons est une exécution de deux sessions en parallèle du protocole. Le protocole implique deux participants : P_1 et P_2. P_1 joue le rôle du leader dans la première session et joue celui d'un membre ordinaire dans la deuxième session. Quant à P_2, il joue le rôle du leader dans la deuxième session et joue celui d'un membre ordinaire dans la première session. Nous avons modélisé donc le protocole par deux rôles de base : un membre ordinaire et un leader. Le rôle du leader n'effectue aucun traitement itératif. En effet, il connaît que le groupe est composé d'un seul membre. Il attend donc la contribution de ce membre, ajoute sa contribution et envoie le résultat à ce membre. Nous nous intéressons aux deux propriétés : le secret de la clef du groupe et l'authentication de la contribution du membre. Nous avons testé cette spécification en considérant deux sessions en parallèle :

```
asokan(Snd,Rcv,m,l,p,f,h)
/\ asokan(Snd,Rcv,l,m,p,f,h)
```

Dans cette spécification, m et l sont les identités respectifs du membre ordinaire et du leader. f et h sont deux fonctions de hachage connues par tous les membres du groupe, et Snd et Rcv sont deux canaux d'envoi et de réception. Avec cette spécification, nous avons obtenu l'attaque décrite par la Figure 3.7.

Dans cette attaque, nous considérons deux sessions en parallèle. La première session implique deux participants $(m, 7)$ qui joue le rôle de leader et $(l, 6)$ qui joue le rôle d'un membre du groupe. Dans la deuxième session, $(m, 3)$ joue le rôle d'un membre ordinaire et $(l, 4)$ joue le

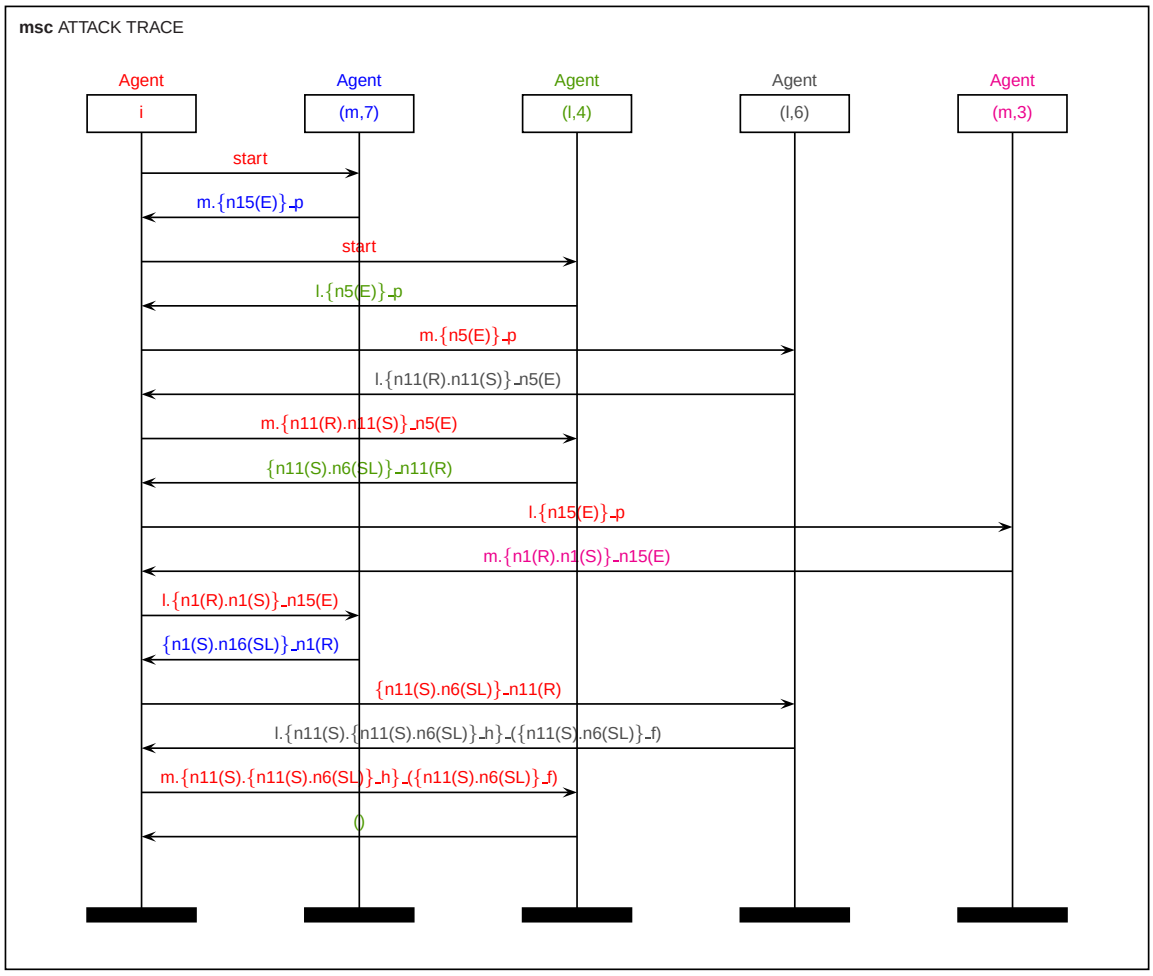


FIG. 3.7 – Attaque d'authentification sur le protocole Asokan-Ginzboorg

rôle du leader. Dans la première session, le membre ordinaire $(l, 6)$ va calculer comme clef de groupe la valeur $f(n11(S).n6(SL))$. Cette valeur était cencée provenir des contributions des deux participants de cette session. Or, le membre $(m, 7)$ n'a pas contribué à cette clef, d'où, l'attaque d'authentification détaillée en Figure 3.7.

La spécification donnant cette attaque ainsi que le résumé de cette attaque sont donnés en Section A.1.2 en Annexe. Notons que pour cette même spécification avec des clefs p différentes, il n'y a pas d'attaque ni de secret, ni d'authentification.

3.5.3 Analyse du protocole GDH

Nous nous intéressons dans cette section à la modélisation du protocole GDH [95]. Nous commençons par les problèmes rencontrés pour modéliser ce protocole. Nous présentons ensuite les spécifications HLPSL des deux scénarii retenus.

Problèmes liées à la modélisation du protocole GDH

Le protocole GDH [95] implique n participants afin de calculer une clef du groupe commune entre eux. Cette clef est calculée à partir des contributions de ces membres qui ont une topologie de chaîne. En effet, en recevant un message (sous forme de concaténation de messages) de son voisin précédant, un membre met sa contribution en exposant de chaque composante du message reçu. Il envoie ensuite le résultat à son voisin suivant. Ce traitement est valable pour tous les membres du groupe mis à part le premier et le dernier membre. Le premier participant déclenche la génération de la clef. Il envoie sa contribution en exposant d'une base α au voisin suivant. Quant au dernier membre, en recevant le message attendu, il considère la dernière composante pour calculer sa clef du groupe. Il va ensuite mettre sa contribtuion en exposant de chacune des autres composantes. Il diffuse le résultat à tout le groupe.

D'après cette description, il existe trois types de membres dans le protocole. Ainsi, nous considérons trois rôles : un rôle d'initiateur (le premier membre), un rôle d'intermédiaire (un membre du groupe mis à part le premier et le dernier) et un rôle final (dernier membre).

Les deux rôles, intermédiaire et final, doivent inclure, chacun un traitement récursif des messages attendus. Pour le rôle intermédiaire, il a besoin de ce traitement pour générer le message à envoyer. En effet, il doit mettre sa contribution en exposant de chacune des composantes du message reçu. Quant au rôle final, il a besoin de distinguer la dernière composante qui lui permet de calculer sa vue de la clef de groupe. Il traite ensuite le reste du message (sans la dernière composante) d'une manière similaire à celle du rôle intermédiaire.

Concernant les deux rôles, intermédiaire et initiateur, ils ont besoin de recevoir un message provenant du dernier membre pour pouvoir calculer leur vue de la clef du groupe. Nous distinguons deux cas :

1. Le dernier membre génère en totalité le message à envoyer en dernière action. Il s'agit donc d'un message composé d'une concaténation de messages sous forme d'exposants. Ce message est par la suite diffusé à tous les membres du groupe. Ceux-ci récupèrent la composante qui les intéresse de ce message selon leur position dans le groupe. Ainsi, pour les deux rôles, intermédiaire et initiateur, ils ont besoin d'effectuer un traitement récursif pour pouvoir récupérer, à partir de la liste reçue, l'information utile pour la génération de la clef.

2. Le dernier membre ne génère pas le message à composer en totalité avant de l'envoyer. En effet, il met sa contribution en exposant de chaque composante du message reçu, et envoie le résultat sans attendre la construction de la totalité du message (la concaténation). Ceci est possible puisque le message à envoyer est une concaténation de composantes. Ainsi, l'envoi de ce message ou l'envoi des composantes revient au même.

Pour modéliser le traitement récursif, nous avons opté pour les transitions sans évènements, i.e. sans les actions *receive-send*.

Première modélisation du protocole GDH

La première variante du protocole est représentée par trois rôles : un rôle initiateur, un rôle intermédiaire et un rôle final dont les actions suivent la description du paragraphe précédant. Dans cette variante, pour la dernière action du rôle final, nous considérons le deuxième cas expliqué dans le paragraphe précédant, i.e. tout au long du traitement du message reçu (une concaténation des messages), le rôle final exponentie chaque composant de ce message (mis à part la dernière) et l'envoie directement sans attendre la construction du message en entier. Quant à la phase d'envoi de contribution, le traitement d'un message reçu par un rôle intermédiaire est spécifié par trois étapes. La première étape est l'initialisation de ce traitement :

```
step2. State=0 /\ Rcv(exp(Xexp1',R1').exp(Xexp11',R11').X2')
   =|>  R2' := new()
        /\ MsgToTreat':= exp(Xexp11',R11').X2'
        /\ SndMsg':= exp(exp(Xexp1',R1'),R2')
        /\ State':=1
```

Dans cette étape, la variable $MsgToTreat$ contient à chaque fois ce qu'il reste du message à traiter. La variable $SndMsg$ contient le message que le membre est en train de composer et qu'il va envoyer à la fin de ce traitement au membre suivant. Cette étape exprime que, en recevant un message formé d'une concaténation de messages sous forme d'exposants, un membre génère sa contribution (R_2), traite la première composante et affecte le résultat à la variable $SndMsg$. Il met à jour aussi la variable $MsgToTreat$ en lui affectant le message reçu privé de de sa première composante.

La deuxième étape de cette première phase est une boucle qui sert à analyser récursivement le message à traiter, i.e. la valeur de la variable $MsgToTreat$. Cette étape est spécifiée comme suit :

```
step4. State=1 /\ MsgToTreat = exp(Xexpi1',Ri1').exp(Xexpi11',Ri11').Xi'
   =|> SndMsg' := SndMsg.exp(exp(Xexpi1',Ri1'),R2)
       /\   MsgToTreat':= exp(Xexpi11',Ri11').Xi'
       /\ State':=1
```

La troisième étape est la condition d'arrêt de l'analyse du message à traiter et donc l'envoi du message composé :

```
step3. State=1 /\ MsgToTreat = exp(Xexpi1',Ri1')
   =|> SndMsg' := SndMsg.exp(Xexpi1',Ri1').exp(exp(Xexpi1',Ri1'),R2)
       /\ Snd(SndMsg')
       /\ State':=2
```

Concernant les actions du rôle final, sa dernière action consiste à un envoi de chaque composante du message reçu traitée (en mettant sa contribution en exposant de cette composante), sans attendre de générer le message (la concaténation de ces messages traités) en totalité.

En analysant cette spécification avec l'outil AVISPA, et en considérant la propriété du secret de la clef du groupe, nous avons trouvé une attaque qui est illustrée par la Figure 3.8. L'exécution considérée est celle du protocole GDH à trois participants : $(m_1, 3)$ jouant le rôle

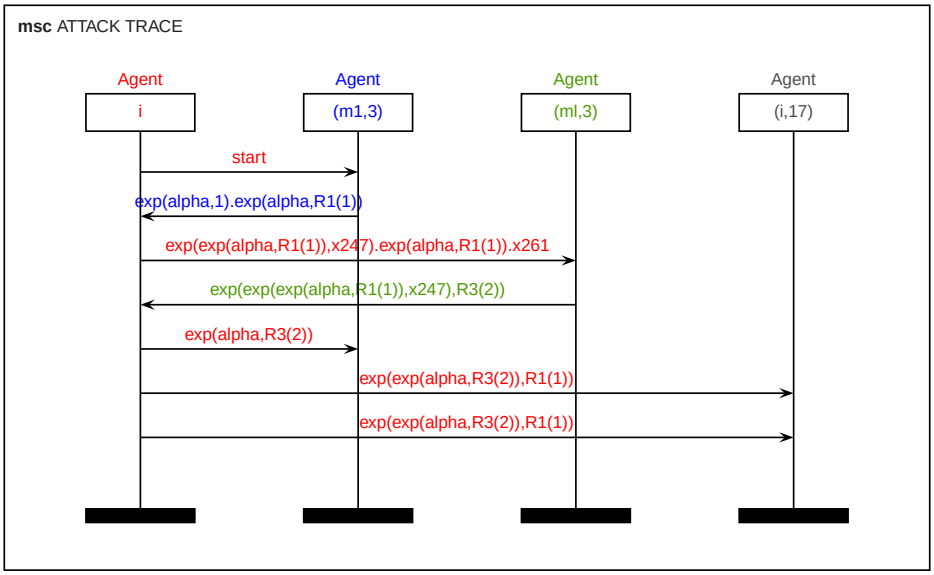


FIG. 3.8 – Attaque de secret sur le protocole GDH

de l'initiateur, $(m_l, 3)$ jouant le rôle final, et $(i, 17)$ jouant le rôle de l'intermédiaire. Dans cette attaque, l'intrus utilise la deuxième composante du message envoyé par le participant $(m, 3)$, i.e. le message $exp(alpha, R1(1))$. Le message qu'il va envoyer au rôle final $(ml, 3)$ sera le message $exp(alpha, R1(1)).exp(alpha, R1(1))$. Ainsi, le leader calculera comme clef de groupe $exp(exp(alpha, R1(1)), R3(3))$ à partir de la deuxième composante du message reçu. Il va aussi envoyer la deuxième composante $exp(alpha, R1(1))$ du message reçu en la mettant en exposant $R3(3)$, ce qui donne la clef du groupe qu'il vient de calculer. Ainsi, la propriété de secret de la clef du groupe pour cette exécution n'est pas satisfaite. Les détails de cette spécification ainsi que l'attaque correspondante sont données en Section A.2.1 en Annexe.

Deuxième modélisation du protocole GDH

Dans cette deuxième variante du protocole GDH, la dernière action du rôle final est de concaténer les composantes traitées du message reçu afin afin d'envoyer le résultat en totalité. Pour générer leurs vues de la clef du groupe, les deux autres rôles, i.e. initial et intermédiaire reçoivent ce dernier message, doivent connaître leurs positions dans le groupe pour pouvoir extraire les composantes qui les intéressent. Le rôle initial extrait la première composante du message reçu. Quant au rôle intermédiaire, il a besoin de parcourir la liste des composantes afin d'extraire celle qui est relative à sa position dans groupe. Cette position est donnée en paramètre à ce rôle par la valeur de *Pos*.

Des variables intermédiaires pour représenter la position courante sont aussi utilisées. Par exemple, la récursivité au niveau du traitement du message pour savoir quelle composante utilisée est modélisée comme suit :

```
step7. State=3 /\ not(CurrentPos=Pos) /\ MsgNeeded=(exp(Xxx',R').Xx')
   =|> State':=3
```

```
/\ CurrentPos' := s(CurrentPos)
/\ MsgNeeded' := Xx'
```

Dans cette action, on vérifie si la première composante n'est pas celle qui correspond au participant en question. Ceci est assuré par le test d'inégalité entre la position de ce participant (Pos) et la position courante du traitement ($CurrentPos$). Dans ce cas, on réitère ce test avec la nouvelle valeur de la variable $MsgNeeded$ qui contiendra l'ancienne valeur privée de sa première composante.

La spécification globale est donnée en Section A.2.2 en Annexe. Nous avons testé cette spécification avec un seul rôle intermédiaire en donnant comme valeur de position courante 2. La vérification de cette spécification a donné une attaque à deux participants : un rôle final et un rôle initiateur. Cette attaque est trouvée avec CL-AtSe. Il s'agit de la même attaque trouvée par OFMC dans la première variante de modélisation. Dans cette attaque, l'intrus récupère le premier message α,α^{R_1} de $m1$ (rôle initiateur) et au lieu de l'envoyer à ml (rôle final) tel qu'il est, il le change avec α^{R_1},α^{R_1}. Le participant, en recevant ce message, va envoyer en clair le message $\alpha^{R_1 R_3}$ qui est déjà la vue de la clef de groupe du participant ml.

3.5.4 Analyse du protocole Tanaka-Sato

Nous avons spécifié le protocole de Tanaka-Sato [97]. Ce protocole est un protocole de gestion de clefs de groupe mettant en cause un serveur et un ensemble de participants ordinaires. Il est composé de quatre sous-protocoles destinés aux requêtes join, leave, send, receive pour modéliser les cas respectivement d'adhésion de groupe, de sortie de groupe, d'envoi et de réception de message. La clef du groupe est distribuée à chaque fois que le serveur reçoit une requête. Quand le serveur reçoit une requête join d'un certain participant, il génère une clef individuelle à ce participant, servant à la communication entre lui et ce participant. Il génère aussi une nouvelle clef du groupe et envoie les deux clefs générées au participant demandeur, toutes les deux encryptées par la clef à long terme du participant en question. À la réception d'une requête leave d'un certain participant, le serveur acquitte l'opération de sortie de ce participant et génère une nouvelle clef de groupe mais ne la distribue pas. En voulant envoyer un message, un participant envoie une requête au serveur pour obtenir la clef courante du groupe. Cette requête contient le numéro de la clef de groupe détenue par le participants. Le serveur vérifie que ce participant fait bien partie du groupe et envoie une liste des clefs de groupe depuis la version de la clef obtenue par le participant en question jusqu'à la version actuelle de cette clef. Le cas send est aussi applicable au cas read.

Nous avons considéré deux variantes considérées de ce protocole que nous présentons ci-après. Nous donnons ensuite les modélisations adoptées ainsi que les résultats obtenus.

Deux scénarii pour la vérification de Tanaka-Sato Pour la vérification du protocole de Tanaka-Sato [97], nous avons distingué deux variantes de ce protocole. La première variante du protocole consiste à considérer le contexte de deux participants. Le premier participant va jouer le rôle du serveur et le deuxième participant jouera le rôle d'un **éventuel** membre du groupe. Le protocole est alors constitué de deux rôles :

1. un rôle de leader ou serveur qui est en état d'attente des différentes requêtes des clients, i.e. des requêtes join, leave, send ou receive ;
2. un rôle d'un membre du groupe décrivant les différentes actions que peut faire un membre ordinaire (requêtes join, leave, send et read). Ces différentes requêtes sont conditionnées

par le fait qu'un participant ne faisant pas partie du groupe ne peut pas directement envoyer des requêtes send, read ou leave.

Dans une deuxième variante du protocole, nous avons considéré le cas d'un agent qui ne fait pas partie d'un groupe existant. Ce groupe est déjà constitué d'un membre et d'un serveur. Le protocole est alors décrit par trois rôles :

1. un rôle de serveur,
2. un rôle d'un membre du groupe pouvant faire différentes requêtes (leave, send et read),
3. un rôle d'un agent qui veut joindre le groupe et par la suite change de statut (création interne d'un nouveau rôle membre avec de nouveaux paramètres) et devient membre du groupe.

Spécification et résultats Pour les deux variantes considérées, les actions des membres ordinaires du groupe ou celles des agents voulant être membres peuvent être définies sans ambiguïté. En effet, pour le rôle d'un participant qui n'est pas encore membre du groupe, il doit tout d'abord être accepté par le serveur. Ainsi, à son état initial, il ne peut faire qu'envoyer une requête join au serveur. Une fois qu'il est accepté, et donc qu'il a reçu sa clef individuelle ainsi que la clef du groupe, il peut passer à un état où il a le choix de procéder à différentes requêtes read, send ou leave. Pour le rôle d'un membre ordinaire du groupe, dès son état initial, il peut déjà choisir parmi les requêtes send, read ou leave.

En ce qui concerne le rôle du serveur, il a besoin de gérer son groupe dynamiquement. En effet, il doit être au courant de la composition de son groupe et des informations liées à ce groupe et doit être capable de répondre aux différentes requêtes qu'il reçoit. Par exemple, à la réception d'une requête join d'un participant, le serveur va chercher la clef commune entre lui et ce participant pour pouvoir déchiffrer le message reçu et va tester si ce participant ne fait pas déjà partie du groupe. Le serveur doit donc disposer de la liste des clefs communes entre lui et les éventuels membres du groupe. Le rôle du serveur gère ensuite différentes autres listes pour sauvegarder les informations relatives au statut du groupe. Il va donc avoir besoin de deux listes : une pour représenter les membres actuels du groupe, et une autre pour sauvegarder les clefs individuelles qu'il a générées pour les différents membres du groupe.

La propriété considérée pour les deux variantes est le secret de la clef du groupe. En effet, que ce soit au départ ou à chaque modification de la structure du groupe (le cas des requêtes join), la nouvelle clef du groupe doit être connue seulement par l'ensemble des membres actuels du groupe. La vérification des deux spécifications, données en Section A.3 en Annexe, relatives aux deux scénarii considérés par AtSe n'a pas abouti. Après quatre jours d'exécution, il n'y avait toujours pas de résultat.

3.5.5 Analyse du protocole Iolus

Nous considérons dans cette section le protocole Iolus [71]. Il s'agit d'un protocole de gestion de clefs de groupe mettant en cause un serveur et un ensemble de participants ordinaires. Il suit le même fonctionnement que le protocole Tanaka-Sato [97] avec quelques différences : principalement, le protocole Iolus distribue la clef de groupe à chaque fois qu'elle est modifiée alors que le protocole de Tanaka-Sato la distribue à la demande. Le protocole Iolus est composé de trois sous-protocoles destinés aux requêtes join, leave et send pour modéliser respectivement l'adhésion de groupe, la sortie de groupe et l'envoi d'un message. La clef du groupe est distribuée à chaque fois que le serveur reçoit une requête. La réponse du serveur à une requête join pour le protocole Iolus suit celle du protocole Tanaka-Sato. En outre, le serveur distribue la nouvelle clef

générée suite à cet évènement en l'encryptant par chacune des clefs individuelles des membres du groupe. À la réception d'une requête leave d'un certain participant, le serveur génère une nouvelle clef et l'envoie à tout le groupe en l'encryptant par l'ancienne clef du groupe. À la différence du protocole de Tanaka-Sato où le membre qui veut envoyer sa contribution envoie une requête de demande de la clef, pour le protocole Iolus, un membre envoie directement son message au groupe vu qu'il détient la dernière clef du groupe.

Spécification et résultat

Nous avons spécifié le protocole Iolus [71] en considérant deux rôles de base : le rôle d'un serveur et celui d'un membre du groupe. Un membre du groupe doit être tout d'abord accepté par le serveur. À son état initial, il ne peut envoyer que des requêtes d'adhésion au groupe. Une fois qu'il fait partie du groupe, le membre peut soit effectuer une requête send ou une requête leave. Le serveur est en attente des différentes requêtes des clients (join, leave). Il doit gérer la dynamique de son groupe. Il dispose des clefs à long-terme des membres susceptibles d'adhérer à son groupe. Le rôle du serveur suit aussi l'évolution de son groupe par le biais de deux listes qui représentent l'ensemble des membres du groupe ainsi que la liste des membres et leur clef individuelle. Ces ensembles sont mis à jour de la même manière en cas d'adhésion d'un nouveau membre ou de sortie d'un membre du groupe.

La propriété considérée pour les deux variantes est le secret de la clef du groupe. La modification de la structure du groupe (join ou leave) entraîne une génération d'une nouvelle clef de groupe qui doit être connue seulement par l'ensemble des membres actuels du groupe.

La vérification de la spécification décrite ci-dessus, donnée en Section A.4 en Annexe, a pris du temps (9763.19 secondes) et a donné comme résultat que le protocole est correct.

3.5.6 Analyse d'une architecture de protocoles hiérarchiques

Le protocole de gestion de clefs considéré dans cette section est une instantiation du protocole Hi-KD [86]. Le groupe géré est structuré sous forme d'une hiérarchie de classes de différents niveaux. Pour assurer leurs communications, les membres d'une classe i utilisent une clef commune entre eux, notée TEK_i. Ils ont aussi accès aux communications des classes inférieures, i.e. les classes j tel que $j > i$. En effet, les clefs de ces classes inférieures peuvent être obtenues par rapport à la clef TEK_i puisque $TEK_{i+1} = f(TEK_i)$ où f est une fonction de hachage. Chacune des classes composant le groupe dispose parmi ses membres d'un membre privilégié qui est le chef de cette classe. Le chef de tout le groupe est celui de la première classe.

Un tel groupe hiérarchique est géré par une architecture de protocoles composée de huit protocoles. Le but est tout d'abord d'établir une clef du groupe via un protocole de génération et de distribution de clefs. Ensuite, la communication de ce groupe doit être sécurisée en tenant compte du dynamisme du groupe, i.e. les différentes opérations affectant la structure du groupe. À chaque opération correspond un protocole qui gère l'adaptation de la clef du groupe à l'événement en question. Parmi les événements, nous pouvons citer le renouvellement de la clef du groupe à chaque période de temps fixée, l'adhésion d'un nouveau membre ou la réintégration d'un ancien membre au groupe, l'exclusion d'un membre, la fusion de deux groupes existants. Vu que le groupe est structuré sous forme de classes hiérarchiques, deux autres opérations peuvent avoir lieu : la promotion d'un membre, i.e. la montée en classe de ce membre ou encore la sanction d'un membre, i.e. la descente en classe de ce membre.

L'ensemble de ces protocoles a été décrit formellement, spécifié en HLPSL et vérifié par l'outil AVISPA. Pour chacun de ces protocoles, l'ensemble des participants impliqués, leurs connaissances initiales ainsi que leurs actions ont été décrits. Pour la modélisation en HLPSL,

nous avons choisi quelques instances de cette architecture. En effet, nous avons considéré un nombre fini de classes. Dans la plupart des cas, nous avons considéré la première classe et deux autres classes inférieures. Le nombre de membres dans les classes varie selon le protocole en question. Par exemple, le protocole de génération de clef est formé d'une première phase impliquant ses trois membres de la première classe et d'une deuxième phase de distribution où cette clef sera distribuée aux classes inférieures. Pour le modéliser, nous avons considéré trois rôles correspondant aux trois membres de la première classe et deux autres rôles correspondant à deux classes inférieures à la classe 1 mais elles aussi de niveaux différents. Pour la plupart des protocoles de l'architecture considérée, nous avons considéré deux rôles par classes, un pour le rôle de chef de cette classe et un pour un rôle ordinaire. Pour certains protocoles de cet architecture, nous les avons testés en augmentant le nombre de membres dans les classes mais l'analyse donne le même résultat que celle de la modélisation initiale. Les propriétés considérées sont le secret et l'authentification. En effet, à chaque fois que la clef du groupe est mise à jour, elle doit être secrète. En outre, cette propriété doit tenir compte de l'architecture du groupe, i.e. une clef d'une certaine classe i ne doit pas être connue ni de l'extérieur ni des classes inférieures j tel que $j > i$. L'authentification est surtout considérée dans le cas de génération d'une clef à la première classe. Elle est utilisée pour s'assurer que les composants de la clef proviennent des membres de la première classe.

L'analyse de l'ensemble des protocoles de cette architecture a mené à une attaque de milieu sur le protocole de promotion et une nouvelle version de ce protocole a été introduite. Le détail des différentes vérifications effectuées sur cette architecture est dans [17, 18, 19, 26].

3.5.7 Discussion

Tout au long des tests des différentes sections précédentes, nous avons pu :

- spécifier un protocole paramétré par le nombre de participants (ou un protocole de groupe) d'une manière abstraite en représentant un ensemble/classe de rôles exécutant les mêmes actions par un seul rôle représentant cette classe.
- modéliser les entrelacements entre les différents protocoles d'une même architecture (ou les différents sous-protocoles d'un même protocole) en raisonnant par rapport aux actions que peut faire un certain rôle. Par exemple, le leader est en état d'attente de requêtes join, leave, ... Dès qu'il reçoit une de ces requêtes, il la traite et revient ensuite à son état d'attente. De même, un éventuel membre ordinaire ne peut effectuer au départ qu'une requête join. Une fois qu'il a adhéré le groupe, il a le choix entre les autres types de requêtes.
- simuler le cas d'un traitement itératif à la réception d'un message. Ce cas a été rencontré pour la plupart des protocoles testés. Par exemple, on peut citer le cas d'un leader qui a besoin des clefs d'encryption de tous les membres du groupe pour pouvoir générer le message à envoyer. Nous avons eu recours dans ces cas à un ensemble spécifiant la clef correspondante à chaque participant.

Néanmoins, pour pouvoir traiter correctement ce genre de protocoles, certaines conditions doivent être satisfaites :

- être capable de traiter les actions itératives. Par exemple, un leader/serveur qui attend la réception des contributions de tous les membres du groupe. Nous avons représenté cette situation par la possession du serveur des différents ensembles (pour les contributions, pour les membres et pour les clefs privées). Néanmoins, il reste encore à vérifier le traitement correct des tableaux.
- être capable de traiter les actions récursives. Par exemple, en recevant un message attendu,

un membre doit effectuer un certain traitement récursif de ce message afin de pouvoir générer le message à envoyer. Pour le cas de GDH, nous avons pu exprimer cette récursivité par des transitions sans événements (sans *receive-send*). Ceci était possible vu que le traitement effectué à chaque composante du message traité est le même. Cependant, le traitement d'une composante d'un message reçu peut nécessiter la connaissance de la position du membre auquel est destiné cette composante. Par exemple, dans le cas de A-GDH, le traitement effectué par le dernier membre, consiste à mettre la clef partagée avec un membre donné, en exposant de la composante destinée à ce membre. Dans ce cas, il est nécessaire, pour le dernier membre, de connaître le statut du groupe et les positions des différents membres. Néanmoins, cette possibilité paraît peu pratique puisqu'elle nécessite un modèle concret de la spécification du protocole du groupe.

– être capable de traiter d'autres propriétés, autres que le secret et l'authentification, et qui sont spécifiques aux protocoles de groupe. Par exemple, si nous considérons les protocoles d'accord de clefs, une propriété spécifique intéressante à vérifier serait de tester si la clef est bien le résultat des contributions de tous les membres du groupe.

3.6 Conclusion

Nous avons présenté dans ce chapitre les problèmes que peut rencontrer la vérification d'une certaine classe des protocoles, appelés protocoles de groupe. Nous avons survolé les travaux qui ont été effectué pour la vérification de ce genre de protocoles. Ils sont classés en deux grandes catégories : ceux qui ont menés à des résultats théoriques et ceux qui sont dédiés à la recherche d'éventuelles attaques sur certains protocoles menant ainsi à la découverte de différentes attaques intéressantes. Nous avons également décrit notre première expérimentation de quelques protocoles de gestion de clefs de groupe. Cette expérimentation a été assurée par l'outil de vérification de protocoles cryptographiques AVISPA. Dans ce travail, nous avons mis l'accent sur les problèmes rencontrés lors de la modélisation des protocoles étudiés et les méthodes adoptées pour cette modélisation.

Les prochains chapitres seront dédiés à nos contributions pour le traitement des problèmes introduits dans ce chapitre. Une partie de notre contribution portera sur la validation d'une classe des protocoles de groupe avec un résultat de décidabilité. Une autre partie de notre contribution se focalisera sur la recherche d'éventuelles attaques sur les protocoles de groupe. Nous commençons dans le chapitre suivant par gérer les propriétés de sécurité des protocoles de groupe en ayant comme objectif la falsification de ces protocoles, i.e. la recherche d'attaques.

4

Détection de types d'attaques sur les protocoles de groupe

Sommaire

4.1 Introduction

Nous avons présenté aux Chapitres 2 et 3, les protocoles de groupe (Section 2.2.2) et les problèmes (Section 3.2) que pose leur vérification. En effet, la sécurité des communications au sein de groupes n'est pas nécessairement une extension d'une communication sécurisée entre deux

parties. Elle est beaucoup plus compliquée. Une fois la communication commencée, le groupe peut changer de structure en ajoutant ou en supprimant un ou plusieurs membres. Les propriétés de sécurité sont alors plus élaborées. Ainsi, vu que les besoins en sécurité sont généralement liés aux protocoles, il est très difficile de décrire les propriétés que le protocole doit satisfaire. Cette phase de spécification des propriétés est critique : toute erreur (de compréhension ou de formalisation de ces propriétés) peut engendrer de fausses failles de sécurité ou l'analyse peut manquer des failles réelles. Des langages formels ont servi à définir des propriétés comme l'authentification : un message est accepté par un participant si et seulement si, il l'a effectivement demandé à un second participant, et ce dernier le lui a bien envoyé. La définition et la spécification des propriétés de sécurité est donc un passage obligé très délicat, avant même d'entamer la vérification. La difficulté est d'autant plus grande lorsqu'il faut considérer des propriétés non standards telles que l'accord de clef ou le secret futur ou encore l'indépendance de clefs.

Le but de ce chapitre est d'abord de définir un modèle pour les protocoles de groupe, et plus généralement pour les protocoles contribuants, i.e. les protocoles où les participants contribuent ensemble afin d'atteindre à un but (par exemple, générer une clef de session). Ce modèle permet de décrire un protocole de groupe, d'étudier ses caractéristiques et ses propriétés de sécurité, et donc d'identifier les différents types d'attaques possibles. L'objectif est ensuite de rechercher d'éventuelles attaques sur les protocoles de groupe en considérant les différentes propriétés de sécurité qui les caractérisent.

Nous commençons en Section 4.2 par définir quelques notions utilisés tout au long de ce chapitre. Nous commençons ce chapitre par présenter notre modèle, appelé aussi modèle de services, sous forme d'un système à plusieurs composantes qui interagissent entre elles. Nous allons donc décrire ces différentes composantes ainsi que les caractéristiques de notre modèle. Nous décrivons par la suite le comportement de notre système suite aux évènements que peut subir le groupe (par exemple, adhésion ou sortie de membres), ce qui nous conduit à formaliser les propriétés de sécurité des protocoles considérés. Le modèle de services a été utilisé pour détecter différents types d'attaques de protocoles d'accord de clefs connus défectueux. Une synthèse des résultats de l'application de ce modèle pour ces protocoles est illustrée dans la section 4.5. Les attaques retrouvées utilisent des scénarii d'exécution de protocole bien précis. Cependant, un tel cadre spécifique pour la modélisation de protocoles est généralement plus utile s'il est connecté à une méthode de vérification afin de pouvoir rechercher et découvrir d'éventuelles attaques en partant juste des spécifications des protocoles. Nous décrivons donc, en Section 4.6 notre procédure de recherche d'attaques sur les protocoles de groupe qui combine à la fois le modèle de services présenté en Section 4.3.2 et le modèle de [101] qui se base sur la résolution de contraintes. Nous définissons dans cette section l'entrée de la méthode de recherche d'attaques ainsi que ses différentes étapes et fonctionnalités. Nous notons ici que le but de cette section est d'utiliser un algorithme existant [101, 24] et l'adapter pour pouvoir rechercher des attaques sur les protocoles de groupe en considérant les différentes propriétés qui les caractérisent et qui sont obtenues par le modèle de services.

Par application de cette méthode pour quelques protocoles de groupe tels que le protocole GDH.2 ou encore le protocole Asokan Ginzboorg, nous avons pu trouver quelques nouvelles attaques. Nous avons aussi généralisé les attaques trouvées pour les protocoles GDH.2 et A-GDH.2 pour les rendre valables à n participants. Une synthèse des résultats de l'application de notre méthode pour ces protocoles est illustrée dans la section 4.7.

4.2 Définitions préliminaires

Comme décrit en Section 1.1.1, un protocole cryptographique est donné par une suite d'échange de messages entre différents principaux. Ces messages peuvent contenir des noms des participants, des nonces et des clefs qu'elles soient symétriques, publiques ou privées. Nous appelons *Clefs* les clefs publiques et clefs privées avec la condition que pour toute clef publique k de *Clefs* la clef privée de k notée k^{-1} appartient aussi à $Clefs$. Nous notons aussi *Noms* comme étant les noms de participants, les nonces et les clefs symétriques. Nous désignons aussi par atome du protocole tout élément de *Noms* ou *Clefs*. Nous notons cet ensemble d'atomes du protocole *Atomes*. À ces atomes s'ajoutent les variables que nous utilisons dans le protocole pour désigner les connaissances des principaux. Nous notons l'ensemble des variables d'un protoccole *Var*. Elles représentent du point de vue des participants les sous-termes des messages reçus qu'ils n'ont pas pu décomposer. À partir des atomes et des variables, nous pouvons construire les messages échangés au cours d'une exécution du protocole. Cette construction est faite par les opérateurs de concaténation (notée $\langle \_, \_ \rangle$), de chiffrement publique (noté $\{\_\}^p_\_$), de chiffrement symétrique (noté $\{\_\}^s_\_$) ou de hachage (noté $H(\_)$) ou encore d'exponentiation (notée $exp(\_, \_)$). Nous définissons donc un message (élément de *Messages*) puis un terme (élément de *Termes*) par les deux langages suivants :

$$\begin{array}{lcl} Messages & :: & Atomes \mid \langle Messages, Messages \rangle \mid \{Messages\}^s_{Messages} \mid \\ & & \{Messages\}^p_{Clefs} \mid H(Messages) \mid exp(Messages, Messages) \\ Termes & :: & Atomes \mid Var \mid \langle Termes, Termes \rangle \mid \{Termes\}^s_{Termes} \mid \\ & & \{Termes\}^p_{Clefs} \mid H(Termes) \mid exp(Termes, Termes) \end{array}$$

Les messages sont donc des termes clos. En décomposant les termes, nous pouvons définir l'ensemble de sous-termes d'un terme t, noté $STermes(t)$ récursivement comme suit :

$$\begin{array}{lcll} STermes(t) & = & \{t, t^{-1}\} & \text{avec } t \in Clefs \\ STermes(t) & = & \{t\} & \text{avec } t \in Var \cup Noms \\ STermes(f(u,v)) & = & \{f(u,v)\} \cup STermes(u) \cup STermes(v) & \text{avec } f \in \{\langle \_, \_ \rangle, \{\_\}^p_\_, \{\_\}^s_\_, exp(\_, \_)\} \end{array}$$

Nous désignons par $\leq$ la relation de sous-termes sur $Termes$ (i.e. $u \leq v$ ssi u est un sous-terme de v). Pour pouvoir définir les exécutions des protocoles en dénotant les connaissances des principaux (l'instantiation des variables) et les différents messages échangés entre les participants, nous avons besoin de définir les substitutions et les substitutions compatibles.
Une substitution est une fonction de Var dans $Termes$ qui assigne des termes aux variables. Une substitution close assigne des messages aux variables. L'application de la substitution σ sur un terme t est notée $t\sigma$. Étant données deux substitutions σ et σ', σ' est dite compatible avec σ si $\forall x \in \mathcal{D}om(\sigma)\ \sigma'(x) = \sigma(x)$.

4.3 Présentation du modèle de services

Le but étant de couvrir la modélisation de la plupart des propriétés de sécurité de tels protocoles, nous commençons par décrire le modèle de services dans la section 4.3.2 en terme de composants. Dans la section 4.3.3, nous décrivons les caractéristiques que doivent satisfaire les composantes de ce modèle. Tout au long de cette section, nous illustrons les notions introduites en utilisant un exemple de protocole de génération de clefs : le protocole A-GDH.2, introduit dans la Section 4.3.1.

4.3.1 Exemples de protocoles de groupe : de DH à A-GDH.2

Atenise et al. introduisent dans [8] le protocole d'accord de clefs de groupe A-GDH.2 (*Authenticated Group Diffie Hellman*) comme une extension naturelle de deux protocoles : le protocole GDH.2 décrit en Figure 2.5 et le protocole A-DH (*Authenticated Diffie Hellman*). Ce dernier protocole, introduit aussi dans [8], est la version authentifiée du protocole de base DH illustré par la Figure 4.1 pour assurer l'authentification de clefs. Dans cette figure, r_A et r_B désignent des nonces générés respectivement par A et B.

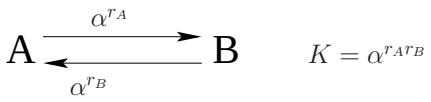


FIG. 4.1 – Le protocole de Diffie-Hellman (DH)

En effet, le besoin d'amélioration de DH a été ressenti après la découverte de l'attaque classique de milieu (*Man In The Middle*), illustrée par la Figure 4.2.

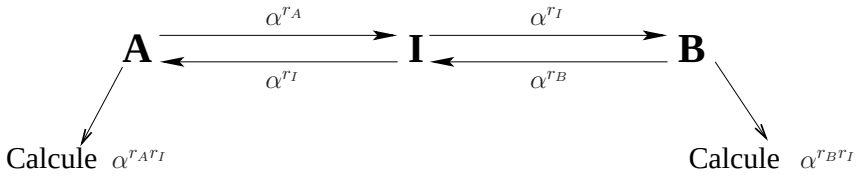


FIG. 4.2 – Attaque d'authentification sur le protocole DH

Dans cette attaque, Alice et Bob essaient d'échanger une clef mais l'intrus intercepte leur message et le remplace par un message qu'il a généré. Alice et Bob arrivent à la fin à calculer chacun une clef qui est partagée par l'intrus. Le protocole A-DH suppose que Alice et Bob ont en commun une clef secrète à long-terme K_{AB}. Il est illustré par la Figure 4.3.

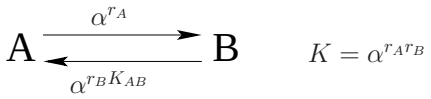


FIG. 4.3 – Le protocole d'accord de clef A-DH

Le protocole A-GDH.2 permet à un groupe de participants $P_1, .., P_n$, disposé sous forme d'un anneau, de partager une clef de session secrète. Nous supposons que α est défini comme pour le protocole DH ou GDH.2. Chaque participant P_i sélectionne une valeur aléatoire secrète r_i. Chaque paire de participants (P_i, P_j) partage aussi une clef secrète à long terme K_{ij}. L'échange

de messages est modélisé comme suit :

$$\begin{array}{l}\text{Phase } i \ (1 \leq i < n)\\ P_i \longrightarrow P_{i+1} \ : \ \left\{\alpha^{\frac{r_1..r_i}{r_j}} \mid j \in [1,i]\right\}, \alpha^{r_1..r_i}\end{array}$$

$$\begin{array}{l}\text{Phase } n\\ P_n \longrightarrow \text{All } P_i \ : \ \left\{\alpha^{\frac{r_1..r_n}{r_j}K_{jn}} \mid j \in [1,n[\right\}\end{array}$$

À la fin de ce protocole, chaque participant P_i calcule la clef de groupe K_G comme suit : $K_G = \alpha^{\frac{r_1..r_n}{r_i}.K_{in}.r_i.\frac{1}{K_{in}}} = \alpha^{r_1..r_n}$

Le protocole, tel qu'il est décrit dans [8], doit satisfaire plusieurs propriétés de sécurité. Parmi ces propriétés, nous trouvons l'authentification de la clef, le secret futur parfait, la résistance aux attaques à clefs connues, la confirmation et l'intégrité de la clef. Toutes ces propriétés et d'autres sont détaillées en Section 2.4. L'exécution du protocole A-GDH.2 pour trois participants P_1, P_2, P_3 peut être décrite en notation Alice-Bob par la séquence de messages suivante :

$$\begin{array}{lll}P_1 & \longrightarrow & P_2 : \ \alpha, \alpha^{r_1}\\ P_2 & \longrightarrow & P_3 : \ \alpha^{r_2}, \alpha^{r_1}, \alpha^{r_1 r_2}\\ P_3 & \longrightarrow & P_1 : \ \alpha^{r_2 r_3 k_{13}}\\ P_3 & \longrightarrow & P_2 : \ \alpha^{r_1 r_3 k_{23}}\end{array}$$

Ce protocole est constitué de quatre messages entre les trois participants. Le troisième et le quatrième messages représentent dans la modélisation initiale du protocole A-GDH.2 le message suivant :

$$P_3 \longrightarrow \text{All } P_i : \alpha^{r_2 r_3 k_{13}}, \alpha^{r_1 r_3 k_{23}}.$$

Cependant, les deux modélisations sont équivalentes puisque nous considérons un intrus de Dolev-Yao. En d'autres termes, du point de vue de l'intrus, la réception d'une concaténation de deux messages en un seul message est équivalente à la réception de deux messages séparés.

Ensuite, le but de ce protocole, comme tout protocole de groupe, est de calculer la clef de groupe. Pour ce faire, chacun des participants doit savoir comment agir suite à la réception de toutes les informations nécessaires au calcul de cette clef de groupe notée $\mathcal{K}_G$. Nous désignons par $\mathcal{A}\mathbf{lg_i}$ l'algorithme effectué par le $i^{ème}$ participant, prenant comme paramètres les informations nécessaires au calcul de la clef et donnant comme résultat la clef du groupe, selon le point de vue de ce participant i.

Pour l'exemple de A-GDH.2, le participant P_1 calcule la clef du groupe en appliquant l'algorithme $\mathcal{A}lg_1(r_1, k_{13}, X_1) = X_1^{r_1/k_{13}}$, où X_1 est la variable désignant le message $\alpha^{r_2 r_3 k_{13}}$. Il est à noter ici que la notion de variable est introduite vu que P_1 ne peut pas vérifier le contenu du message reçu. De même, P_2 calcule $\mathcal{A}lg_2(r_2, k_{23}, X_2) = X_2^{r_2/k_{23}}$, où X_2 est la variable désignant $\alpha^{r_1 r_3 k_{23}}$. Enfin, P_3 calcule $\mathcal{A}lg_3(r_3, X_3) = X_3^{r_3}$, où X_3 désigne le message $\alpha^{r_1 r_2}$. Il est à noter aussi que dans cet exemple, puisqu'il s'agit d'une exécution normale du protocole, i.e. une exécution sans intervention d'un intrus, tous les membres calculent la même clef du groupe ($\mathcal{K}_G = \alpha^{r_1 r_2 r_3}$).

4.3.2 Un modèle de services à base de composantes

Un protocole d'établissement de clefs de groupe, tel que le protocole A-GDH.2 a pour objectif de calculer une clef de session commune en se basant sur des informations secrètes générées par

chacun des participants. Nous constatons que, pour la plupart de ces protocoles, un participant honnête du groupe *contribue* à la clef du groupe en offrant aux autres participants des *services* destinés essentiellement au calcul de cette clef. Ces services sont nécessaires pour le bon calcul de la clef de groupe. De la même manière, un participant a besoin des contributions des autres membres du groupe pour pouvoir calculer la bonne clef du groupe.

Nous constatons aussi que, étant donné que la clef du groupe se base essentiellement sur les services des différents membres du groupe modulo d'autres informations, il s'avère que divulguer un ou plusieurs services à l'intrus peut nuire au bon fonctionnement du protocole et à la satisfaction de propriétés de sécurité voulues pour ce type de protocoles. La définition des propriétés d'un tel type de protocoles peut être alors facilement définie en se basant sur la notion de services. Par exemple, si l'intrus arrive à avoir tous les services nécessaires pour la clef, il arrive sûrement à construire la clef du groupe. Dans ce cas, la propriété de *secret* de la clef du groupe ne serait pas satisfaisable. De la même manière, si l'intrus contribue par ses propres informations à la construction de la clef du groupe, i.e. il existe un service utilisé pour la clef du groupe qui provient de l'intrus, alors l'*intégrité* de la clef du groupe ne serait pas satisfaite. Le modèle que nous proposons dans cette section se base sur ces constatations.

Pour pouvoir décrire les propriétés de sécurité des protocoles d'établissement de clefs de groupe, nous proposons un modèle manipulant les services des participants formant le groupe. Il consiste à définir un système global de plusieurs composantes qui interagissent entre elles. Ainsi, un protocole d'établissement de clefs de groupe peut être vu comme un système représenté par un triplet $<\mathbf{P}, \mathbf{K}, \mathbf{S}>$, avec,

$\mathbf{P}$: ensemble des participants membres du groupe,
$\mathbf{K}$: ensemble des connaissances des participants,
$\mathbf{S}$: ensemble des services.

Nous allons, dans ce qui suit, donner une vue plus approfondie des différents éléments composants les trois ensembles P, K et S.

La première composante de notre système (P) contient les membres du groupe. Nous désignons par $\mathbf{P_i} \in P$ pour $i \in \mathbb{N}$ le $i^{ème}$ participant du groupe. Ceci n'impose pas un ordre pour les participants dans le groupe. D'ailleurs, parmi les études de cas, il existe des protocoles où il n'y a pas d'ordre pour les participants tel que le protocole de Asokan-Ginzboorg [6]. Il existe aussi parmi ces mêmes études de cas, des protocoles où l'ordre est très important dans leur exécution, comme le protocole A-GDH.2, utilisé dans ce chapitre comme étude de cas.

La deuxième composante (K) désigne l'ensemble des connaissances des participants. Nous définissons ici les sous-ensembles formant K. Soit $i \in \mathbb{N}$, alors :

- $\mathbf{K_i} \subseteq K$ est l'ensemble des connaissances privées de P_i. Il s'agit de l'ensemble minimal des connaissances utiles pour construire les services et la clef finale. Il inclut les connaissances privées au départ de la session ainsi que les informations engendrées par P_i au cours du déroulement de la session ;
- $\mathbf{K_{ij}} \subseteq K$ est l'ensemble des connaissances partagées par les agents P_i et P_j. C'est l'ensemble minimal des connaissances partagées et utiles pour la génération de la clef de groupe. Ces informations sont données par la spécification du protocole. Il est à noter que $K_{ij} = K_{ji}$.

La troisième composante (S) désigne l'ensemble de services. Un service selon Pereira et Quisquater [81] est défini comme étant le calcul effectué par un participant honnête durant l'exécution du protocole. L'entrée et le résultat de ce calcul peuvent être interceptés par l'intrus. Plus précisément, un service est une fonction qui à α^x associe α^{xp}. Pour notre cas, notre but est

de pouvoir traiter une large classe de protocoles et non pas uniquement les protocoles à base de Diffie-Helman. Nous définissons alors un service comme suit :

Définition 4.3.2.1 *(Service.)*
Nous désignons par service *toute* contribution *d'un participant pour engendrer la clef de groupe. La contribution d'un participant P_i à un autre agent P_j est toute information engendrée par P_i, utile pour P_j afin de déduire la clef de groupe ou bien de construire une autre information, qui elle, sera utile pour la construction de la clef.*

En s'intéressant plus aux différentes interactions possibles entre un participant P_i et l'ensemble de services S, nous avons défini des sous-ensembles de S. L'ensemble $\mathbf{S_i} \subseteq S$ désigne l'ensemble minimal des services utiles à P_i pour construire la clef de groupe. En plus des services utiles pour engendrer la clef de groupe, on distingue d'autres sous-ensembles de services utilisés pour regrouper les services offerts par chaque agent. Ainsi, nous notons S_{c_i} le sous-ensemble de services auxquels P_i a contribué (directement ou indirectement) par l'utilisation d'informations privées.

$$\mathbf{S_{c_i}} = \{s \in S \mid \exists t; t \in STermes(s), \text{ tel que } t \in K_i\}$$

Exemple 4.3.2.2 *Pour illustrer les différentes définitions ci-dessus, nous revenons à notre exemple A-GDH.2. Si nous décrivons ce protocole dans notre modèle, nous obtenons :*

$$\begin{array}{l} P = \{P_1, P_2, P_3\}, \\ K = \{r_1, r_2, r_3, k_{13}, k_{23}\}, \\ S = \{\alpha^{r_1}, \alpha^{r_2}, \alpha^{r_1 r_2}, \alpha^{r_1 r_3 k_{23}}, \alpha^{r_2 r_3 k_{13}}\} \end{array}$$

Les informations relatives aux agents sont illustrées dans le tableau suivant :

$\mathbf{P_i}$	$\mathbf{S_i}$	$\mathbf{S_{c_i}}$	$\mathbf{K_i}$	$\cup_\mathbf{j}\mathbf{K_{ij}}$
$\mathbf{P_1}$	$\alpha^{r_2 r_3 k_{13}}$	$\alpha^{r_1}, \alpha^{r_1 r_2}, \alpha^{r_1 r_3 k_{13}}$	r_1	k_{13}
$\mathbf{P_2}$	$\alpha^{r_1 r_3 k_{23}}$	$\alpha^{r_2}, \alpha^{r_1 r_2}, \alpha^{r_2 r_3 k_{13}}$	r_2	k_{23}
$\mathbf{P_3}$	$\alpha^{r_1 r_2}$	$\alpha^{r_2 r_3 k_{13}}, \alpha^{r_1 r_3 k_{23}}$	r_3	k_{13}, k_{23}

En résumé, notre modèle est donc basé sur un triplet $<P, K, S>$, décrivant que chaque participant P_i **possède** des informations confidentielles ($\mathcal{K}_i$), **partage** des informations (K_{ij}) avec d'autres participants (par exemple P_j), **a besoin** de connaissances (S_i) pour engendrer la clef de groupe et **contribue** (S_{c_i}) à la génération de la clef aux autres participants du groupe. Ces relations entre les composantes relatives à un participant sont illustrées par la Figure 4.4.

4.3.3 Caractéristiques du modèle de services

Notre système est donc composé de trois ensembles de base P, K, et S, à partir desquels sont construits de nombreux autres ensembles (K_i, K_{ij}, S_i, S_{c_i}). La construction de ces ensembles pour un protocole donné est assez simple, mais ils doivent cependant satisfaire quelques caractéristiques indispensables. Nous décrivons ci-dessous ces caractéristiques, qui concernent essentiellement les interactions entre ces différents ensembles. Ces caractéristiques sont résumées dans la Figure 4.5 et illustrées par l'exemple de A-GDH.2.

Les entités communiquants dans des protocoles d'établissement de clefs sont formellement appelées principaux et sont supposées avoir des noms uniques.

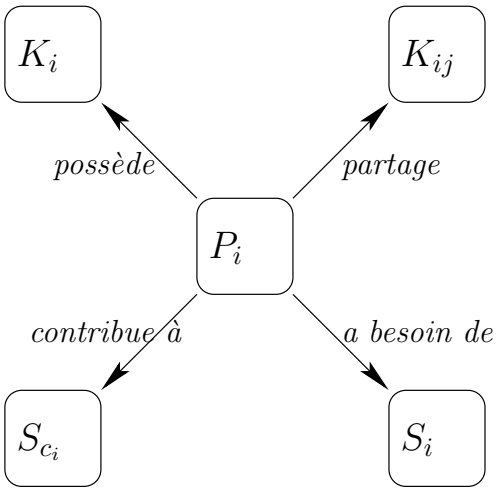


FIG. 4.4 – Composantes liées à P_i

Caractéristique 1 (Unicité des identificateurs des participants) *Les identités des principaux doivent être différentes.*

$$\forall P_i \in P, \forall P_j \in P, (i \neq j) \Longrightarrow (P_i \neq P_j) \ .$$

Caractéristique 2 (Visibilité des connaissances privées) *Les connaissances dites « privées » des agents doivent être réellement confidentielles. Elles ne peuvent pas être partagées :*

$$\forall P_i \in P, \forall P_j \in P, (i \neq j) \Longrightarrow (K_i \cap K_j = \emptyset) \ .$$

$$\forall P_i \in P, \forall t \in K_i, \forall P_j, P_k \in P, t \notin K_{jk} \ .$$

De plus, elles ne peuvent pas être diffusées en clair :

$$\forall P_i \in \mathcal{P}, \forall t \in K_i, t \notin S \ .$$

Caractéristique 3 (Visibilité des connaissances partagées) *Les connaissances partagées entre plusieurs agents sont en fait des connaissances confidentielles vis-à-vis des autres agents. Chaque agent sait avec qui il partage des connaissances :*

$$\forall P_i, P_j \in P, \forall t \in K_{ij}, (\exists P_k, P_l \in P, k \neq i, j \wedge t \in K_{kl}) \Longrightarrow (t \in K_{ik} \cap K_{kj}) \ .$$

Et comme pour les connaissances privées, les connaissances partagées ne doivent pas être transmises en clair :

$$\forall P_i, P_j \in P, \forall t \in K_{ij}, t \notin S \ .$$

Caractéristique 4 (Distinction des services utiles) *Les ensembles de services nécessaires à la construction de la clef de groupe pour deux agents doivent être différenciés par au moins un élément.*

$$\forall P_i, P_j \in A, (i \neq j) \Longrightarrow (S_i \neq S_j) \ .$$

Caractéristique 5 (Indépendance des services utiles) *Les ensembles de services nécessaires à la construction de la clef de groupe pour deux agents ne doivent pas être liés par une relation d'inclusion.*

$$\forall P_i, P_j \in P, (i \neq j) \Longrightarrow (S_i \nsubseteq S_j) \ .$$

Au contraire des protocoles de distribution de clefs, dans les protocoles d'échange de clefs de groupe, aucune partie ne doit être capable de choisir la clef de groupe résultante au nom des autres membres du groupe. Ainsi, les services utiles de chacun des autres participants doivent correspondre à des services fournis par les autres participants :

Caractéristique 6 (Correspondance des services) *Tout élément de S_i doit correspondre à un élément d'un S_{c_k}. Cela signifie que les services nécessaires à un agent P_i pour construire la clef doivent provenir de contributions d'autres agents.*

$$\forall P_i \in P, \forall s \in S_i, \exists P_k \in P, s \in S_{c_k} \ .$$

Les caractéristiques définies précédemment concernent la construction des ensembles de connaissances et de services attachés aux agents. Ces informations sont utilisées par les agents pour construire la clef de groupe $\mathcal{K}_G$. Cette phase de déduction (notée $\models$) de $\mathcal{K}_G$, intervenant à la fin de la session du protocole, doit également être contrôlée. Nous désignons par *déduction* l'application de certaines règles de composition et de décomposition de messages tout en considérant l'hypothèse de chiffrement parfait. Ces règles (qui définissent $\models$) sont données par la Table 4.1.

Règles de Composition	**Règles de Décomposition**
$\models k \ , \ \ k \notin \mathcal{K} \cup \mathcal{A} \cup \mathcal{S}$	
$m_1,\ m_2 \models < m_1, m_2 >$	$< m_1, m_2 > \models m_1 \, ; < m_1, m_2 > \models m_2$
$m,\ k \models \{m\}_k^p$	$\{m\}_k^p,\ inv(k) \models m$
$m,\ inv(k) \models \{m\}_{inv(k)}^p$	$\{m\}_{inv(k)}^p,\ k \models m$
$m,\ b \models \{m\}_b^s$	$\{m\}_b^s,\ b \models m$
$x,\ y \models x.y$	$x.y,\ y^{-1} \models x \, ; x.y^{-1},\ y \models x$
$y \models y^{-1}$	$y^{-1} \models y$
$t,\ \alpha \models \alpha^t$	

TAB. 4.1 – Définition de $\models$

Ces règles présentent les règles de déduction d'un intrus de Dolev-Yao [45]. Dans ce modèle, l'intrus a le contrôle total de la communication sur le réseau. Il peut donc intercepter, enregistrer, modifier, envoyer, chiffrer et déchiffrer des messages s'il possède la bonne clef (de chiffrement ou de déchiffrement). Il a la possibilité aussi d'envoyer des faux messages en se faisant passer pour un autre participant honnête.

Toutes les règles de la Table 4.1 sont modulo l'associativité et la commutativité des exposants. Elles s'accompagnent également de simplifications, liées aux propriétés de ces opérateurs algébriques.

Caractéristique 7 (Déduction de la même clef de groupe) *Pour un agent P_i, la clef de groupe est engendrée en appliquant l'algorithme $\mathcal{A}lg_i$ aux connaissances privées et partagées, ainsi qu'aux services nécessaires de cet agent.*

$$\mathcal{A}lg_i(K_i, \cup_{P_k \in P} K_{ik}, S_i) \models \mathcal{K}_G \ .$$

Tous les membres du groupe doivent déduire la même clef. L'application de l'algorithme doit donc donner le même résultat pour tous les membres.

$$\forall P_i \in P, \mathcal{A}lg_i(K_i, \cup_{P_k \in P} K_{ik}, S_i) \models \mathcal{K}_G \ .$$

Caractéristique 8 (Services minimaux) *Un groupe de services est dit minimal si l'accomplissement de l'objectif auquel il est destiné nécessite la participation de tous les éléments du groupe.*
Dans le cadre de notre système, toutes les contributions doivent être utilisées pour la génération de la clef de groupe. Un participant P_i ne peut pas se limiter à un sous-ensemble S'_i de S_i pour construire la clef.

$$\forall P_i \in P, \nexists S'_i \subset S_i, \mathcal{A}lg_i(K_i, \cup_{P_k \in P} K_{ik}, S'_i) \models \mathcal{K}_G \ .$$

Un participant P_i doit donc avoir recours à tout l'ensemble S_i en tenant compte de ses connaissances privées et partagées. Une conséquence est la propriété suivante : aucun élément de S_i ne peut être déduit des autres éléments de cet ensemble.

$$\forall P_i \in P, \forall s \in S_i, (S_i \backslash \{s\}) \cup K_i \cup_j K_{ij} \not\models s \ .$$

La structure du système ainsi que ses caractéristiques sont illustrées par la figure 4.5.

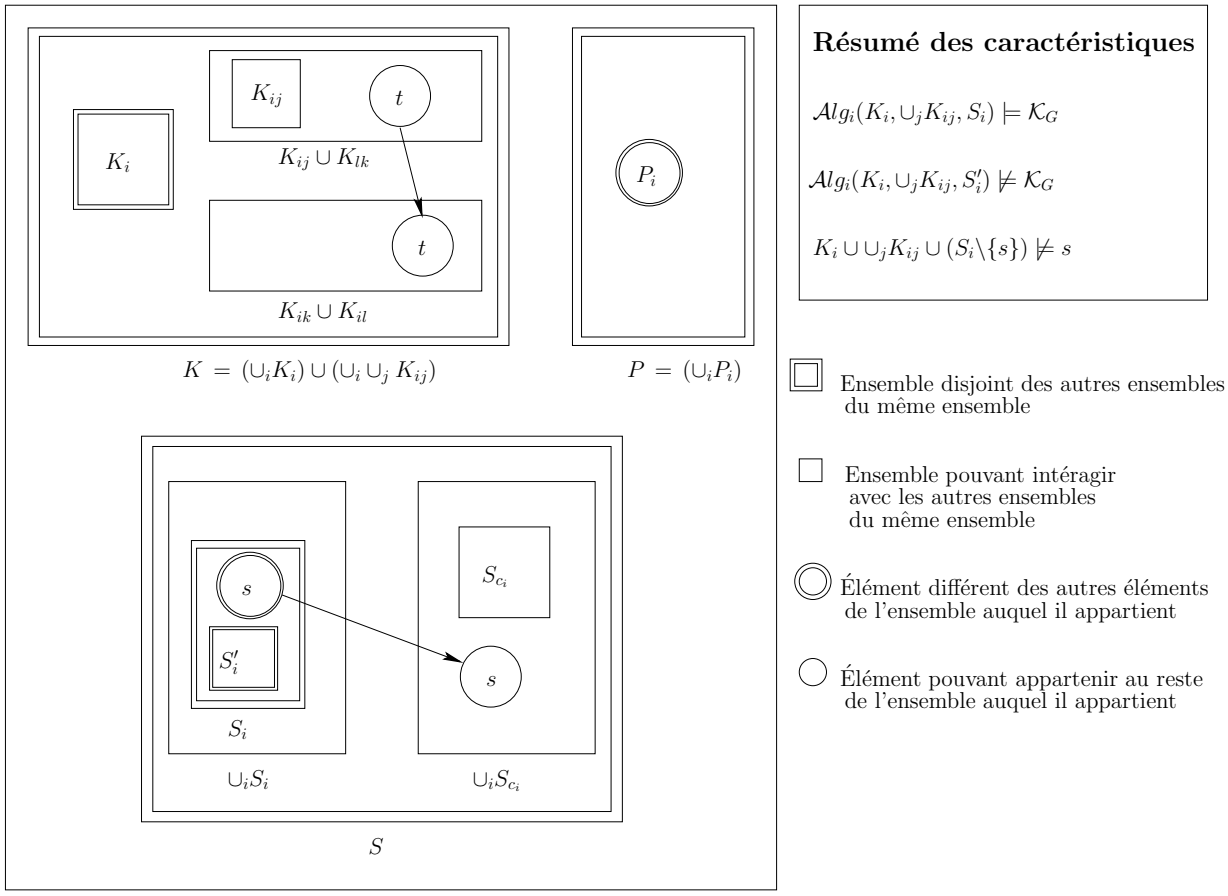


FIG. 4.5 – Modèle de services et ses caractéristiques

Exemple 4.3.3.1 *Nous revenons à notre exemple de A-GDH.2 pour lequel, nous étudions une à une les caractéristiques définies précédemment.*

1. *L'unicité des identificateurs des agents est vérifiée.*
2. *La caractéristique de visibilité des connaissances privées est vérifiée. En effet, les ensembles des connaissances privées sont disjoints :*

$$\{r_1\} \cap \{r_2\} \cap \{r_3\} = \emptyset$$

Et d'autre part,

$$r_1 \notin \mathcal{S}, r_2 \notin \mathcal{S}, \text{ et } r_3 \notin \mathcal{S}$$

3. *La caractéristique de visibilité des connaissances partagées est vérifiée puisque les trois ensembles K_{12}, K_{13} et K_{23} sont disjoints. En plus, $k_{13} \notin S \wedge k_{23} \notin S$.*
4. *Les propriétés de distinction de services utiles et d'indépendance des services utiles sont vérifiées car :*

$$S_1 \cap S_2 = S_1 \cap S_3 = S_2 \cap S_3 = \emptyset$$

5. *La caractéristique de correspondance de services est vérifiée pour les trois membres P_1, P_2 et P_3. En effet,*
   – pour P_1, $S_1 \subset S_{c_2}$ et $S_1 \subset S_{c_3}$;

– *pour* P_2, $S_2 \subset S_{c_3}$ *et* $S_2 \subset S_{c_1}$;
– *pour* P_3, $S_3 \subset S_{c_1}$ *et* $S_3 \subset S_{c_2}$.

6. *La caractéristique de déduction de la même clef de groupe est vérifiée. Tout d'abord, la clef du groupe est calculée par chacun des membres comme suit :*

$$\begin{aligned} \mathcal{A}lg_1(r_1, k_{13}, \alpha^{r_2 r_3 k_{13}}) = \alpha^{r_2 r_3 k_{13} * r_1 / k_{13}} &\models \alpha^{r_1 r_2 r_3} \\ \mathcal{A}lg_2(r_2, k_{23}, \alpha^{r_1 r_3 k_{23}}) = \alpha^{r_1 r_3 k_{23} * r_2 / k_{23}} &\models \alpha^{r_1 r_2 r_3} \\ \mathcal{A}lg_3(r_3, \emptyset, \alpha^{r_1 r_2}) = \alpha^{r_1 r_2 * r_3} &\models \alpha^{r_1 r_2 r_3} \end{aligned}$$

Tous les membres du groupe déduisent donc la même clef ($\alpha^{r_1 r_2 r_3}$).

7. *La caractéristique de services minimaux est vérifiée. En effet, pour les trois agents, l'ensemble* S_i *est l'ensemble minimal à partir duquel on arrive à engendrer la clef du groupe.*

4.3.4 Caractéristiques relatives à la dynamique du groupe

Le modèle introduit dans la section 4.3.2 décrit un *état stable* du système qui représente une vue d'un état courant du groupe. Il s'agit d'une instance d'une génération de clef d'un protocole de groupe. Le processus de cette génération est traduit par les relations entre les différents ensembles de l'état du système. Néanmoins, un groupe de membres n'est en général pas stable au cours du temps. En effet, le groupe peut évoluer par changement de sa composition en ajoutant ou en supprimant un ou plusieurs membres. Le groupe peut aussi changer de structure. Cet évènement peut se présenter dans le cas où le groupe est présenté sous forme d'architecture hiérarchique. Par exemple, un groupe de pompiers, ayant des grades différents, voulant communiquer. Ce groupe est donc formé de différentes classes hiérarchiques, où chaque classe ne peut communiquer qu'avec les classes de même niveau ou d'un niveau inférieur. Le changement de cette structure peut avoir lieu par exemple par une descente de classe ou bien une montée. Pour plus de détails sur un tel exemple, se référer à la Section 2.2.2. Le système n'a donc pas intérêt à se limiter à un seul état. Puisqu'un groupe est modélisé par un système à trois composantes P, K et S, l'évolution du système suivant l'évolution du groupe se traduit par la modification d'au moins une de ses trois composantes. Par exemple, le changement de composition du groupe tel que l'ajout ou la suppression d'un membre modifie l'ensemble P. De même, le changement de structure du groupe tel que la montée ou la descente de classe fait modifier à la fois les deux ensembles K et S.

Afin de prendre en compte cette dynamique du système, nous allons étendre le modèle proposé dans la section 4.3.2 en introduisant une modélisation du temps. Soit T l'ensemble (infini) des entiers naturels qui va modéliser le temps. Le modèle proposé auparavant ne représente que l'état observé à un instant τ.

$$GP^{\tau} = <P, K, S>^{\tau} = <P^{\tau}, K^{\tau}, S^{\tau}> \ .$$

Le système global est donc donné par :

$$GP^{T} = \{GP^{\tau}, \tau \in T\} \ .$$

L'ensemble T est muni d'un ordre noté "<". Soient τ et τ' deux instants de T. L'influence du temps dans le modèle du système, défini dans la section 4.3.2, peut être alors vue comme suit :
Si $\tau < \tau'$ alors GP^{τ} est un état qui *précède* temporellement $GP^{\tau'}$.
Si $\tau > \tau'$ alors GP^{τ} est un état qui *suit* temporellement $GP^{\tau'}$.
Si $\tau = \tau'$ alors il s'agit du *même* état GP^{τ}.

Notion d'événement. Le passage d'un état GP^{τ} à un autre état $GP^{\tau'}$ se fait suite à des *événements*. On désigne par événement toute opération capable de modifier l'un des constituants de l'état du système. Dans le cadre de protocoles de groupe, un événement peut être vu comme un changement de la composition du groupe dû à une opération d'ajout ou de suppression d'un membre du groupe, ou un changement de structure du groupe dû par exemple à une montée ou descente de classe ou une mise à jour des données du groupe tel que l'expiration d'un délai (par exemple pour un rafraîchissement de clé). Dans le cadre de notre modèle, un événement peut affecter l'ensemble des agents P (ajout ou élimination de participants), ou le sous-ensemble des connaissances privées d'au moins un agent ou le sous-ensemble des connaissances partagées d'au moins un agent. Cette propriété peut s'exprimer comme suit :

$$<P, K, S>^{\tau} \longrightarrow_{event} <P, K, S>^{\tau'} \ .$$

si on a :

$$(P^{\tau} \neq P^{\tau'}) \vee (P^{\tau} = P^{\tau'} \wedge \exists P_i \in P, K_i^{\tau} \neq K_i^{\tau'}) \vee (P^{\tau} = P^{\tau'} \wedge \exists P_i, P_j \in P, K_{ij}^{\tau} \neq K_{ij}^{\tau'}) \ .$$

ce qui revient à la condition suivante :

$$(P^{\tau} \neq P^{\tau'}) \vee (P^{\tau} = P^{\tau'} \wedge K^{\tau} \neq K^{\tau'}) \ .$$

Pour illustrer ces conditions, nous revenons à nos petits exemples. Le changement de la composition du groupe entraîne un changement de l'ensemble des participants et donc la composante P change. Le changement de la structure du groupe par des opérations de type montée ou descente de classe n'affecte pas l'ensemble des membres du groupe mais modifie les connaissances partagées du ou des agents concernés. Par exemple, un membre qui monte une classe aura comme connaissances, en plus de ses connaissances courantes, des connaissances lui permettant de communiquer avec les membres de la classe à laquelle il est affecté. Une mise à jour des données du groupe telle que le rafraîchissement de clés est un événement qui modifie les connaissances privées (comme par exemple le changement de nombre aléatoire r_i).

4.4 Formalisation des propriétés de sécurité

Nous avons présenté en Section 2.4 les propriétés de sécurité pour les protocoles d'établissement de clefs de groupe. Nous allons dans cette section nous intéresser aux propriétés des protocoles d'accord de clefs et les classer selon leurs dépendances ou indépendances vis à vis du temps. Nous distinguons donc deux types de propriétés de sécurité : les propriétés indépendantes du temps et celles dépendantes du temps. Pour pouvoir définir les attaques, i.e, la non-vérification des propriétés, nous avons besoin tout d'abord de modéliser l'intrus. Nous considérons l'intrus de Dolev-Yao, i.e. il a le contrôle de tout le réseau. Les déductions de l'intrus sont représentées par $\models$, définie par la Table 4.1.

Concrètement, dans un protocole d'établissement de clefs de groupe, un intrus est soit un participant non officiel du protocole et donc qui n'appartient pas au groupe, soit un participant officiel (ancien ou actuel membre du groupe) utilisant sa position avantageuse pour perpétrer des actions malhonnêtes. Dans ce type de protocoles, l'intrus peut essayer plusieurs stratégies afin de nuire au fonctionnement du protocole. Il peut essayer de déduire la clef de session en utilisant les informations interceptées (intrus passif). Il peut aussi participer secrètement à un protocole initié par un participant avec un autre et influencer le déroulement du protocole en modifiant des messages afin de pouvoir déduire la clef de session. Il peut aussi initier une ou

plusieurs exécutions du protocole et les combiner en entrelaçant les messages d'un participant avec un autre afin de se faire passer pour un des membres du groupe. Il peut aussi tromper un participant en se faisant passer pour un autre membre avec qui il partage une clef. Dans notre modèle, les actions de l'intrus sont interprétées selon deux cas. Dans le premier cas, un intrus peut engendrer des actions laissant le système dans son état stable telles que l'accès aux différents services offerts ($\bigcup_{P_i \in P} S_{c_i}$) ou l'interception ou la modification de messages échangés. Dans le deuxième cas, l'intrus peut aussi engendrer des actions provoquant un changement d'état comme l'envoi d'un message d'initialisation de session de protocole.

4.4.1 Cas statique

Il s'agit des propriétés de sécurité liées à un état stable du système. Dans cette section, nous modélisons les propriétés des protocoles d'échange de clefs de groupe qui sont indépendantes de temps, à savoir les propriétés d'authentification implicite de clef, de secret de la clef du groupe, de confirmation de clef et d'intégrité définies en Section 2.4.1 du Chapitre 2.

Un protocole vérifie la propriété d'*authentification implicite* de la clef (IKA) d'après la Définition 2.4.2.1 si, à la fin de la session, chaque participant est assuré qu'aucun élément externe ne peut acquérir sa vue de la clef de groupe.

Dans notre modèle défini dans la Section 4.3.2, la vue d'un participant P_i est représentée par l'application de l'algorithme $\mathcal{A}lg_i$ ayant pour paramètres les informations nécessaires pour engendrer la clef du groupe ($K_i \cup S_i \cup_j K_{ij}$).

Un intrus (généralement noté I) peut avoir une telle vue s'il peut la déduire grâce aux différentes informations qu'il possède, à savoir ses connaissances privées, ses éventuelles connaissances partagées avec les autres participants et les différents services offerts par les participants du groupe ($K_I \cup_j K_{Ij} \cup_{P_k \in P} S_{c_k}$).

La propriété d'authentification implicite de la clef peut être alors définie dans notre modèle comme suit :

Propriété 1 (Authentification implicite de la clef) *La propriété d'authentification implicite de la clef pour un participant P_i, différent de l'intrus I, est exprimée comme suit :*

$$K_I \cup_j K_{Ij} \cup_{P_k \in P} S_{c_k} \not\models \mathcal{A}lg_i(K_i, \cup_j K_{ij}, S_i) \ .$$

Dans un état du système, seuls les membres du groupe peuvent engendrer la clef partagée. Ceci est assuré pour les protocoles satisfaisant la propriété de secret de la clef de groupe définie par la Définition 2.4.1.1. Un agent externe I peut violer cette propriété en déduisant la clef du groupe à partir de ses connaissances privées, ses éventuelles connaissances partagées avec les autres participants et des connaissances acquises, i.e les différents services offerts par les participants du groupe ($K_I \cup_j K_{Ij} \cup_{P_k \in P} S_{c_k}$).

La propriété de secret de la clef de groupe peut être alors définie dans notre modèle comme suit :

Propriété 2 (Secret de la clef) *La violation de la propriété de secret de la clef de groupe est définie par :*

$$K_I \cup_j K_{Ij} \cup_{P_k \in P} S_{c_k} \models \mathcal{K}_G \ .$$

Étant donné que dans un contexte de groupe, chacun des participants a une vue de la clef du groupe qu'il déduit à partir de ses informations y compris ses connaissances privées, ses

connaissances partagées avec les autres participants du groupe et ses services utiles, la propriété de secret de la clef du groupe peut alors être définie comme suit :

$$\forall P_i \in P, K_I \cup_j K_{Ij} \cup_{P_k \in P} S_{c_k} \not\models \mathcal{A}lg_i(K_i, \cup_j K_{ij}, S_i) \ .$$

Il est à noter que cette définition est semblable à celle de la propriété 1 sauf qu'ici, le raisonnement se fait sur tous les membres le groupe ($\forall P_i \in P$) au lieu de raisonner par rapport à un seul élément (P_i) qui est le cas pour l'authentification.

Dans le contexte de protocoles d'accord de clefs de groupe, il existe une propriété servant à convaincre une partie du groupe que les autres membres arrivent à engendrer la clef partagée : la *confirmation* de la clef (Définition 2.4.2.2).

Comme la seule possibilité pour un participant du groupe pour pouvoir engendrer la clef est d'avoir les services nécessaires, la confirmation de clef revient alors à la confirmation de ces services.

Pour qu'un participant P_i confirme avoir engendré la clef, il doit communiquer à chaque émetteur P_j d'un service utile s à la génération de la clef de P_i , qu'il a bien reçu cette contribution s. Cette notion de confirmation deux à deux peut être alors définie comme suit :

Propriété 3 (Confirmation de la clef) *En considérant les membres deux à deux, un agent P_i confirme avoir reçu le service correspondant à un agent P_j afin de l'utiliser pour la génération de sa vue de la clef de groupe si :*

$$\exists s \in S_i | s \in S_{c_j} \ .$$

La confirmation de la clef de groupe peut être alors définie par rapport à tout le groupe en considérant chaque paire de participants de ce groupe.

La propriété d'*intégrité*, telle qu'elle est définie dans la Définition 2.4.2.3, permet de vérifier que tous les agents contribuent à la clef de groupe et que toute partie extérieure au groupe ne doit pas participer à la génération de la clef partagée.

La première condition est vérifiée si l'ensemble des services S_i de chaque participant P_i résulte des contributions de tous les autres participants du groupe. Il s'agit donc de la confirmation de la clef définie par la Propriété 3.

La deuxième condition est vérifiée si aucun élément de S_i ne provient d'une partie extérieure au groupe. Sinon, il y aurait nécessairement un intrus qui a contribué aux services utiles de P_i et donc qui a contribué à la génération de la vue de la clef du groupe de ce participant.

Propriété 4 (Intégrité) *L'intégrité de la clef est donc définie, pour tous les membres du groupe, d'une part par la confirmation des services utiles pour engendrer la clef en considérant chaque paire possible de participants du groupe :*

$$\forall P_i, P_j \in P \ (i \neq j), \exists s \in S_i | s \in S_{c_j} \ .$$

et d'autre part, par la correspondance des services :

$$\forall P_i \in P, \forall s \in S_i, \exists P_j \in P | s \in S_{c_j} \ .$$

Comme un intrus I pourrait ne participer qu'en partie à la conception d'un service nécessaire pour un participant, la propriété traitée exprime aussi qu'aucun sous-terme des services utiles ne peut appartenir aux connaissances de l'intrus :

$$\forall P_i \in P, \forall s \in S_i, \forall t \text{ sous-terme de } s, t \notin K_I \ .$$

4.4.2 Cas dynamique

Certaines propriétés de sécurité ont un lien fort avec la notion de temps. L'une des plus importantes est l'*indépendance de clefs de groupe* définie dans la Définition 2.4.3.8. Elle garantit qu'aucun sous-ensemble de clefs ne peut être utilisé pour découvrir un autre sous-ensemble de clefs qui lui est disjoint. Il s'agit en fait de la combinaison de deux propriétés de sécurité : le secret futur défini par la Définition 2.4.3.6 et le secret passé défini par la Définition 2.4.3.7, tous les deux présentés en Section 2.4.3.

Secret futur La propriété de *secret futur* garantit qu'un adversaire passif qui connaît un certain nombre d'anciennes clefs de groupe ne peut pas découvrir une clef de groupe plus récente. Soit par exemple une séquence de clefs de groupe $\{K_0, \ldots, K_m\}$. La propriété de secret futur exprime que, connaissant un sous-ensemble $\{K_0, K_1, \ldots, K_i\}$, un intrus passif ne peut pas découvrir une clef de groupe K_j telle que $j > i$.

Dans notre modèle, pour pouvoir définir une telle propriété, il est nécessaire de détailler encore plus la définition de la propriété de *Secret de la clef de groupe*. En effet, le secret d'une clef de groupe est relatif à un instant bien précis correspondant à l'état courant du groupe. Cette propriété est donc la suivante : à tout instant, il est impossible (du point de vue calculatoire) à l'intrus de découvrir la clef de groupe à partir des informations de l'instant considéré. Formellement, cette propriété est définie par :

$$\forall \tau \in T, K_I^\tau \cup_{P_j \in P^\tau} K_{Ij}^\tau \cup_{P_k \in P^\tau} S_{c_k}^\tau \not\models \mathcal{K}_G^\tau \ .$$

Nous pouvons maintenant définir la propriété de secret futur comme suit :

Propriété 5 (Secret futur) *La propriété de secret futur est spécifiée dans notre modèle ainsi :*

$$\forall \tau_i, \tau_j \in T, i < j, \{\mathcal{K}_G^{\tau_1}, \mathcal{K}_G^{\tau_2}, \ldots, \mathcal{K}_G^{\tau_i}\} \not\models \mathcal{K}_G^{\tau_j} \ .$$

avec, $\forall k, l, \tau_k < \tau_l$ *si* $k < l$.

Cette définition de la propriété de secret futur peut être encore raffinée, en décrivant les attaques qui peuvent être détectées. Nous définissons deux variantes selon l'information connue conduisant à l'attaque. La première variante traite le cas de compromis d'un service d'une session passée.

Propriété 6 (Protection contre une attaque par service connu) *Un protocole est dit vulnérable à ce type d'attaque si la connaissance d'un service d'une session passée permet d'avoir la vue d'un agent pour la clef d'une session future. En supposant que l'état du système est à l'instant* τ*, il* ***n'y a pas*** *d'attaque par service connu* ***si*** *:*

$$\forall \tau' \in T, \tau' < \tau, \forall P_i \in P^\tau, S^{\tau'} \cup K_I^\tau \cup_j K_{Ij}^\tau \cup_{P_k \in P} S_{c_k}^\tau \not\models \mathcal{A}lg_i(K_i^\tau, \cup_j K_{ij}^\tau, S_i^\tau) \ .$$

La seconde variante traite le cas de compromis d'une clef d'une session passée.

Propriété 7 (Protection contre une attaque par clef connue) *Un protocole est dit vulnérable à ce type d'attaque si la connaissance d'une clef à long terme d'un agent d'une session passée permet d'obtenir la vue d'un agent pour la clef d'une session future. En supposant que l'état du système est à l'instant* τ*, il* ***n'y pas*** *d'attaque par connaissance de clef* ***si*** *:*

$$\forall \tau' \in T, \tau' < \tau, \forall P_i \in P^\tau, K^{\tau'} \cup K_I^\tau \cup_j K_{Ij}^\tau \cup_{P_k \in P} S_{c_k}^\tau \not\models \mathcal{A}lg_i(K_i^\tau, \cup_j K_{ij}^\tau, S_i^\tau) \ .$$

Secret passé Cette deuxième propriété garantit qu'un intrus passif connaissant un sous-ensemble de clefs de groupe ordonnées $\{K_i, K_{i+1}, \ldots, K_j\}$ ne peut pas découvrir une clef de groupe précédente K_l et ceci pour tous l, i, j tels que $l < i < j$.

Propriété 8 (Secret passé) *Cette propriété s'énonce ainsi :*

$$\forall \tau_i, \tau_j, \tau_l \in T, \tau_l < \tau_i < \tau_j, \{\mathcal{K}_G^{\tau_i}, \mathcal{K}_G^{\tau_{i+1}}, \ldots, \mathcal{K}_G^{\tau_j}\} \not\models \mathcal{K}_G^{\tau_l} \ .$$

4.5 Détection de types d'attaques

Après avoir défini notre modèle dans la Section 4.3 et formalisé quelques propriétés de sécurité des protocoles de groupe dans la Section 4.4, nous présentons dans cette section les résultats obtenus par application de notre modèle pour quelques protocoles connus défectueux. Nous détaillons en particulier le résultat obtenu pour notre étude de cas : le protocole A-GDH.2.

Pour chacun de ces protocoles étudiés, nous nous sommes basé sur des scénarii d'attaques de ces protocoles pour les modéliser dans notre système. Cette modélisation nous a permis de tester une à une les caractéristiques ainsi que les propriétés définies auparavant afin d'identifier les types d'attaques auxquels ces protocoles seraient vulnérables.

4.5.1 Analyse du protocole A-GDH.2

Il est à noter que même à partir d'une seule phase d'un scénario d'attaque (voir Figure 4.6), comme c'est le cas de A-GDH.2, nous avons pu détecter plusieurs propriétés de sécurité non vérifiées. Le scénario que nous considérons est une exécution du protocole à quatre participants où l'intrus joue le rôle du troisième participant.

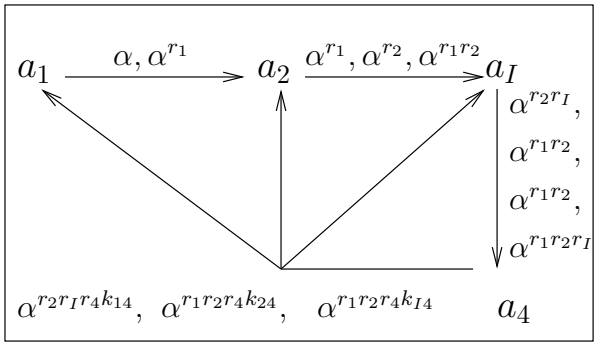


FIG. 4.6 – Attaque sur A-GDH.2

La modélisation de cette phase dans notre modèle nous donne :

$\mathbf{P_i}$	$\mathbf{S_i}$	$\mathbf{S_{c_i}}$	$\mathbf{K_i}$	$\cup_\mathbf{j} \mathbf{K_{ij}}$
$\mathbf{a_1}$	$\alpha^{r_2 r_I r_4 k_{14}}$	$\alpha^{r_1}, \alpha^{r_1 r_2}, \alpha^{r_1 r_2 r_I}, \alpha^{r_1 r_2 r_4 k_{I4}}, \alpha^{r_1 r_2 r_4 k_{24}}$	r_1	k_{14}
$\mathbf{a_2}$	$\alpha^{r_1 r_2 r_4 k_{24}}$	$\alpha^{r_2}, \alpha^{r_1 r_2}, \alpha^{r_2 r_I}, \alpha^{r_1 r_2 r_I}, \alpha^{r_1 r_2 r_4 k_{I4}}, \alpha^{r_1 r_2 r_4 k_{24}}, \alpha^{r_2 r_I r_4 k_{14}}$	r_2	k_{24}
$\mathbf{a_4}$	$\alpha^{r_1 r_2 r_I}$	$\alpha^{r_1 r_2 r_4 k_{I4}}, \alpha^{r_1 r_2 r_4 k_{24}}, \alpha^{r_2 r_I r_4 k_{14}}$	r_4	k_{14}, k_{24}, k_{I4}

À partir de ces informations, nous déduisons que la caractéristique *Déduction de la même clef de groupe* n'est pas satisfaite. En effet, les vues de clef de a_1 et de a_2 ne convergent pas vers la

même clef puisque :

$$\begin{aligned} \mathcal{A}lg_1(r_1, k_{14}, \alpha^{r_2 r_I r_4 k_{14}}) = \alpha^{(r_2 r_I r_4 k_{14}) * r_1 / k_{14}} &\models \alpha^{r_1 r_2 r_I r_4}, \\ \mathcal{A}lg_2(r_2, k_{24}, \alpha^{r_1 r_2 r_4 k_{24}}) = \alpha^{(r_1 r_2 r_4 k_{24}) * r_2 / k_{24}} &\models \alpha^{r_1 r_2 r_4 r_2}, \\ \mathcal{A}lg_4(r_4, \emptyset, \alpha^{r_1 r_2 r_I}) = \alpha^{(r_1 r_2 r_I) * r_4} &\models \alpha^{r_1 r_2 r_I r_4} \end{aligned}$$

Si nous étudions maintenant la propriété de sécurité *authentification implicite*, l'intrus (I) peut avoir la même vue que a_4. Grâce aux informations qu'il possède (r_I et k_{I4}), l'intrus utilise le service offert $\alpha^{r_1 r_2 r_4 k_{I4}}$ pour déduire la vue de a_4 comme suit :

$$\alpha^{(r_1 r_2 r_4 k_{I4}) * r_I / k_{I4}} \models \alpha^{r_1 r_2 r_4 r_I} = \mathcal{A}lg_4(r_4, \emptyset, \alpha^{r_1 r_2 r_I})$$

4.5.2 Synthèse des protocoles analysés

Nous présentons dans cette section les protocoles étudiés ainsi que les résultats trouvés. Les protocoles analysés sont des protocoles connus défectueux. Le choix des scénarii de ces protocoles a été effectué en se basant sur des attaques qui ont été déjà trouvées. Le but était de retrouver ces attaques. Néanmoins, nous avons choisi des protocoles ayant des propriétés de sécurité non satisfaites différentes. En effet, ayant tous le même objectif, à savoir l'établissement d'une clef de session, les propriétés non satisfaites par ces protocoles sont différentes de l'un à l'autre. Parmi ces propriétés violées, nous trouvons des propriétés simples telles que le secret ou l'authentification. Nous avons également trouvé des attaques pour les propriétés de confirmation de la clef, d'intégrité, de secret futur avec ses deux variétés d'attaques par service connu ou par clef connue.

Dans la Table 4.2, nous précisons les protocoles que nous avons étudié, leurs références ainsi que les références des scénarii d'attaques utilisés, et les types d'attaques trouvés pour ces scénarii.

4.6 Recherche d'attaques sur les protocoles de groupe

Nous présentons dans cette section notre procédure pour rechercher d'éventuelles attaques sur des protocoles de groupe. Nous commençons par introduire, en Section 4.6.1, un exemple de protocole d'accord de clef : le protocole de Asokan-Ginzboorg. Cet exemple sera utilisé pour illustrer les différentes notions introduites pour la procédure de recherche d'attaques. Nous précisons dans la Section 4.6.2 les données de notre méthode d'analyse. Ces données comprennent en résumé une spécification du protocole en règles de réception-envoi et une spécification de la ou des propriétés de sécurité à vérifier. Elle résulte de la modélisation du protocole en modèle de services. Nous donnons ensuite dans la Section 4.6.3 notre algorithme de recherche d'attaques qui se base essentiellement sur la combinaison du modèle [101] et notre modèle de services. Nous détaillons dans cette section les différentes variables et fonctions utilisées dans l'algorithme ainsi que les différentes étapes de cet algorithme. Nous présentons après, en Section 4.6.4 la notion d'arbre de contraintes qui permet la gestion des contraintes et la construction d'une éventuelle attaque pour un protocole étudié.

4.6.1 Exemple de protocole de groupe : Asokan-Ginzboorg

Le protocole de Asokan Ginzboorg [6] décrit l'établissement d'une clef de session entre un leader (P_n) et un nombre $n-1$ arbitraire de participants (P_i avec $1 \leq i \leq n-1$). Le protocole suppose qu'un mot de passe de groupe p a été choisi et diffusé à tous les participants formant le

Protocole	**Référence protocole**	**Référence attaque**	**Caractérisique(s) falsifiée(s)**	**Propriété(s) falsifiée(s)**
A-GDH.2	[80]	[81]	– déduction de la même clef de groupe.	– authentification implicite, – secret de la clef, – confirmation de la clef, – intégrité.
SA-GDH.2	[8]	[81]	– déduction de la même clef de groupe.	– authentification implicite, – secret de la clef, – secret futur (attaque par service connu).
Asokan-Ginzboorg	[6]	[94]	– déduction de la même clef de groupe.	– authentification implicite.
Bresson-Chevassut-Essiari-Pointcheval	[22]	[74]	– correspondance de services.	– secret futur (attaque par service connu et attaque par clef connue).

TAB. 4.2 – Synthèse des protocoles étudiés

groupe. On suppose aussi que tous ces participants connaissent deux fonctions de hachage h et f.

Le leader commence l'exécution du protocole en générant une clef publique (e) et en l'envoyant au reste du groupe. Cette clef sera utilisée après pour encrypter les informations nécessaires pour la génération de la clef du groupe. À la réception de cette clef, chacun des participants génère deux informations. La première est une clef symétrique r_i qui va être utilisée pour la communication des différentes informations nécessaires au calcul de la clef de groupe entre le leader et ce participant. Quant à la deuxième information, c'est la contribution du participant en question à la clef du groupe, un nonce s_i. Une fois ces deux informations générées, le participant les envoie encryptées par la clef e au leader. Après la récolte de toutes les contributions des différents participants, le leader génère sa propre contribution s_n à la clef du groupe. Il la concatène avec toutes les contributions qu'il vient de recevoir et envoie le résultat à chacun des participants en l'encryptant par la clef de chacun de ces participants. En recevant ce message, chacun des participants peut calculer la clef de groupe qui doit être égale à $f(s_1, \ldots, s_n)$. Pour confirmer son calcul, chaque participant envoie un message de confirmation au leader en encryptant le message envoyé par la nouvelle clef de session.

La séquence de messages échangés tout au long de ce protocole est la suivante :

$1.P_n \longrightarrow$ Tout : $a_n, \{e\}_p$
$2.P_i \longrightarrow P_n$: $a_i, \{r_i, s_i\}_e$, i=1,..,n-1
$3.P_n \longrightarrow P_i$: $\{\{s_j, \text{j=1,..,n}\}\}_{r_i}$, i=1,..,n-1
$4.P_i \longrightarrow P_n$: $a_i, \{s_i, h(s_1, .., s_n)\}_k$ pour $i, k = f(s_1, .., s_n)$.

Dans cet échange de messages, e désigne une clef publique générée par le leader ; a_i pour $i = 1..n$ sont les identités des différents participants P_i ; s_i pour $i = 1..n$ sont des nonces représentant les contributions pour la clef de groupe des différents participants y compris le leader ; r_i pour $i = 1..(n-1)$ représentent les clefs symétriques des participants (sauf le leader).

4.6.2 Données de la méthode

Notre méthode a pour objectif de rechercher d'éventuelles attaques sur les protocoles de groupe. Comme tout protocole de communication, un protocole de groupe peut être vu comme un échange de messages entre différents participants. Cet échange est généralement décrit par un ensemble d'actions exécutées par chacun des participants dans une exécution normale du protocole. Nous désignons par exécution normale du protocole une exécution où l'intrus n'intervient pas. Nous commençons donc par définir le modèle de protocole([101]) que nous suivons.

Modèle du protocole

Nous définissons dans cette section le modèle de protocole suivi, i.e. ce qu'on appelle un pas de protocole, un protocole, un ordre d'exécution et une exécution.

Définition 4.6.2.1 *Pas de protocole.*
Un pas de protocole est une paire de termes $\mathcal{R}$ et $\mathcal{S}$, noté $\mathcal{R} \Rightarrow \mathcal{S}$.

L'exécution de ce pas de protocole par un participant A, modulo une substitution close σ décrivant les connaissances de A avant le pas considéré, se traduit par ces deux faits :

1. la réception par A d'un message t tel que $\exists \sigma'$ compatible avec σ, telle que $t = \mathcal{R}\sigma'$;
2. la réponse de A en envoyant le message $\mathcal{S}\sigma'$.

σ' traduit la mise à jour des connaissances de A suite à l'acquisition de nouvelles connaissances à la réception de t.

Une fois qu'un pas de protocole est défini, nous pouvons alors définir un protocole en réunissant un ensemble de pas de protocoles et des connaissances initiales de l'intrus :

Définition 4.6.2.2 *Protocole.*
Un protocole P est un triplet $(\{\mathcal{R}_p \Rightarrow \mathcal{S}_p \mid p \in \mathcal{P}\}_\mathcal{P}, <_\mathcal{P}, CI_0)$ avec,
- *$\mathcal{P}$ un ensemble d'indices permettant d'identifier tous les pas de protocole ;*
- *$<_\mathcal{P}$ est un ordre partiel sur $\mathcal{P}$;*
- *$\mathcal{R}_p \Rightarrow \mathcal{S}_p$ est un pas de protocole. Parmi les messages utilisés dans les pas d'un protocole, il existe deux messages fixés qui sont $Init$ et End qui sont utilisés pour initier et fermer une session ;*
- *CI_0 est un ensemble fini de messages représentant l'ensemble de connaissances initiales de l'intrus.*

et avec la condition suivante :

$$(C) \ : \ \forall p \in \mathcal{P}, \ \forall x \in Var(\mathcal{S}_p), \ \exists p' \leq_\mathcal{P} p \text{ tel que } x \in Var(\mathcal{R}_{p'})$$

où $\leq_\mathcal{P}$ est la clôture réflexive de $<_\mathcal{P}$ (i.e $p \leq_\mathcal{P} p'$ ssi $p <_\mathcal{P} p'$ ou $p = p'$).

La condition C exprime que toute variable utilisée dans un envoi de message par un participant A à un pas p doit exister dans un message reçu d'un pas antérieur à p et donc A connaît sa valeur au moment de l'envoi. Intuitivement, un participant n'a pas le droit d'envoyer un message contenant une variable s'il ne connaît pas sa valeur et donc s'il n'y a pas de pas précédent où cette variable a été instanciée.

L'ordre partiel $<_{\mathcal{P}}$ sert à décrire les différents ordonnancements possibles entre les pas des participants. Le but de notre procédure de recherche d'attaques est de rechercher d'éventuelles attaques pour n'importe quel comportement de participants. En effet, un participant est libre de choisir d'exécuter un pas p avant un autre pas p' à condition que $p' \not<_{\mathcal{P}} p$. Ainsi, quelque soit l'ordre des exécutions de pas choisi par chacun des participants (à condition de respecter l'ordre partiel du protocole), notre procédure doit chercher des exécutions susceptibles de correspondre à des attaques. Nous définissons donc dans ce qui suit la notion d'ordre d'exécution et la notion d'exécution.

Définition 4.6.2.3 *Ordre d'exécution.*
Soit $P = (\{\mathcal{R}_p \Rightarrow \mathcal{S}_p | p \in \mathcal{P}\}_{\mathcal{P}}, <_{\mathcal{P}}, CI_0)$ *un protocole. Un ordre d'exécution* $<_e$ *sur* $\mathcal{P}$ *de domaine* $\mathcal{P}'$ *est un ordre total sur* $\mathcal{P}' \subseteq \mathcal{P}$ *compatible avec* $<_{\mathcal{P}}$ *et stable par prédécesseur, i.e. pour tous* $p_1 \in \mathcal{P}$ *et* $p_2 \in \mathcal{P}'$*, si* $p_1 <_{\mathcal{P}} p_2$ *alors si* $p_1 \in \mathcal{P}'$*,* $p_1 <_e p_2$*.*

Un ordre d'exécution total $<_e$ correspondant à une exécution (totale) du protocole est un ordre total sur $\mathcal{P}$ compatible avec $<_{\mathcal{P}}$, i.e. si $p_1 <_{\mathcal{P}} p_2$ pour $\{p_1, p_2\} \in \mathcal{P}$ alors $p_1 <_e p_2$.

Définition 4.6.2.4 *Exécution.*
Une exécution d'un protocole P *est un couple* (π, σ) *où* π *désigne un ordre d'exécution et* σ *une substitution close sur* Var*.*

Retour à l'exemple de Asokan Ginzboorg

Ayant testé notre méthode sur différents scénarii du protocole de Asokan Ginzboorg, nous avons trouvé d'intéressants résultats pour le cas de deux sessions parallèles.

Ainsi, nous considérons dans ce paragraphe une modélisation du scénario trouvé. Nous avons donc deux sessions exécutées en parallèle. Dans la première session, nous avons deux participants P_1 et P_2 avec le participant P_1 qui joue le rôle de leader et le participant P_2 qui joue le rôle d'un membre normal du groupe. Dans la deuxième session, nous avons toujours les deux participants P_1 et P_2 sauf que cette fois les rôles sont inter-changés, i.e., P_1 est un membre normal et P_2 est le leader.

Tout au long des exemples de ce chapitre, nous adoptons les notations suivantes :

- pas_{ijk} représente la j-ième étape du protocole jouée par le i-ième participant dans la k-ième session.
- Les termes écrits en minuscules présentent les informations qui sont connues par les participants englobant des connaissances initiales ou encore des informations acquises durant l'exécution.
- Les termes écrits en lettres majuscules représentent les variables. Ils peuvent être instanciés par n'importe quelle valeur à condition qu'elle soit de même type.
- s_{ij} représente la contribution à la clef de groupe du i-ième participant dans la j-ième session.
- Alg_{ij} est la vue de la clef du groupe du participant i durant la session j.

Les deux sessions sont décrites par les pas suivants :

$$
\begin{array}{lrcl}
\mathbf{pas_{111}}: & Init & \longrightarrow & a_1, \{e_1\}_p \\
\mathbf{pas_{121}}: & X_1, \{X_2, X_3\}_{e_1} & \longrightarrow & \{X_3, s_{11}\}_{X_2} \\
\mathbf{pas_{131}}: & X_1, \{X_3, h(X_3, s_{11})\}_{f(X_3,s_{11})} & \longrightarrow & End \\
 & & & \\
\mathbf{pas_{211}}: & X_4, \{X_5\}_p & \longrightarrow & a_2, \{r_2, s_{21}\}_{X_5} \\
\mathbf{pas_{221}}: & \{X_6, X_7\}_{r_2} & \longrightarrow & a_2, \{s_{21}, h(X_6, X_7)\}_{f(X_6,X_7)} \\
 & & & \\
 & & & \\
\mathbf{pas_{212}}: & Init & \longrightarrow & a_2, \{e_2\}_p \\
\mathbf{pas_{222}}: & X_8, \{X_9, X_{10}\}_{e_2} & \longrightarrow & \{X_{10}, s_{22}\}_{X_9} \\
\mathbf{pas_{232}}: & X_8, \{X_{10}, h(X_{10}, s_{22})\}_{f(X_{10},s_{22})} & \longrightarrow & End \\
 & & & \\
\mathbf{pas_{112}}: & X_{11}, \{X_{12}\}_p & \longrightarrow & a_1, \{r_1, s_{12}\}_{X_{12}} \\
\mathbf{pas_{122}}: & \{X_{13}, X_{14}\}_{r_1} & \longrightarrow & a_1, \{s_{12}, h(X_{13}, X_{14})\}_{f(X_{13},X_{14})}
\end{array}
$$

Dans la première session, P_1 a comme vue de la clef de groupe $Alg_{11} = f(X_3, s_{11})$ et P_2 a comme vue de la clef de groupe $Alg_{21} = f(X_6, X_7)$. D'une manière similaire, dans la deuxième session, P_1 a comme vue de la clef de groupe $Alg_{12} = f(X_{13}, X_{14})$ et P_2 a comme vue de la clef de groupe $Alg_{22} = f(X_{10}, s_{22})$.

L'ensemble $\mathcal{P} = \{pas_{111}, pas_{121}, pas_{131}, pas_{211}, pas_{221}, pas_{212}, pas_{222}, pas_{232}, pas_{112}, pas_{122}\}$ des étapes est ordonné par l'ordre partiel $<_{\mathcal{P}}$ suivant :

$$
\begin{array}{l}
pas_{111} <_{\mathcal{P}} pas_{121} <_{\mathcal{P}} pas_{131} \\
pas_{211} <_{\mathcal{P}} pas_{221} \\
pas_{212} <_{\mathcal{P}} pas_{222} <_{\mathcal{P}} pas_{232} \\
pas_{112} <_{\mathcal{P}} pas_{122}.
\end{array}
$$

Une modélisation du protocole dans le modèle de services

À partir de l'ensemble de pas $\{\mathcal{R} \Rightarrow \mathcal{S}\}_{\mathcal{P}}$ de différents participants, nous modélisons le protocole dans le modèle de services décrit dans le chapitre 4 afin de construire l'ensemble de contraintes relatives à la violation de la (des) propriété(s) à vérifier. Dans cette modélisation, nous spécifions pour chaque participant, les services utiles, les services contribuants, les connaissances privées ainsi que les connaissances partagées.

Puisque les règles formant ces pas contiennent des variables, les informations nécessaires pour la modélisation du protocole dans le modèle de services contiennent aussi des variables. Ces variables sont le sujet d'un ensemble de contraintes spécifiant la violation de la ou des propriétés à vérifier. L'ensemble de contraintes est un système d'équations mettant en relation les variables avec d'autres variables ou les variables avec des constantes. Nous notons $SC_{\mathcal{P}}$ cet ensemble de contraintes qui modélisent la violation de la (des) propriété(s).

Par exemple, si nous considérons la propriété d'accord de clefs qui découle de la caractéristique de déduction de la même clef de groupe décrite en Section 4.3.3, cette propriété est décrite dans le modèle de services comme suit :

$$\forall P_i, P_j \mid i \neq j \; \mathcal{A}lg_i(K_i, \cup_{P_k \in P} K_{ik}, S_i) = \mathcal{A}lg_j(K_j, \cup_{P_k \in P} K_{jk}, S_j).$$

En considérant la spécification du protocole Asokan Ginzboorg, soient

– K_{il} les connaissances privées du participant P_i lors de la l-ième session,

- S_{il} les services nécessaires du participant P_i lors de la l-ième session,
- K_{ijl} les connaissances partagées entre les deux participants P_i et P_i durant le l ème session,
- Alg_{il} la vue de la clef de groupe du i-ième participant durant le l-ième session.

La modélisation de ces deux sessions dans le modèle de services donne le résultat suivant :

$\mathbf{P_i}$	$\mathbf{S_i}$	$\mathbf{S_{c_i}}$	$\mathbf{K_i}$	$\cup_\mathbf{j}\mathbf{K_{ij}}$
P_1	X_3, $\{X_{13}, X_{14}\}_{r_1}$	$\{e_1\}_p, \{X_3, s_{11}\}_{X_2}$, $\{r_2, s_{12}\}_{X_{12}}, \{s_{12}, h(X_{13}, X_{14})\}_{f(X_{13},X_{14})}$	a_1, e_1, s_{11}, r_1, s_{12}	h, f, p
P_2	$\{X_6, X_7\}_{r_2}$ X_{10}	$\{r_1, s_{21}\}_{X_7}, \{s_{21}, h(X_6, X_7)\}_{f(X_6,X_7)}$ $\{e_2\}_p, \{X_{10}, s_{22}\}_{X_9}$,	a_2, r_2, s_{21}, e_2, s_{22}	h, f, p

La propriété d'accord de clefs pour chacune des sessions considérée est alors modélisées comme suit :

$$\mathcal{A}lg_{11}(K_{11}, K_{121}, S_{11}) = \mathcal{A}lg_{21}(K_{21}, K_{211}, S_{21}).$$
et
$$\mathcal{A}lg_{12}(K_{12}, K_{122}, S_{12}) = \mathcal{A}lg_{22}(K_{22}, K_{212}, S_{22}).$$

Ce qui se traduit par :

$$f(X_3, s_{11}) = f(X_6, X_7)$$
et
$$f(X_{13}, X_{14}) = f(X_{10}, s_{22}).$$

ce qui donne :

$$X_3 = X_6 \text{ et } X_7 = s_{11}$$
et
$$X_{13} = X_{10} \text{ et } X_{14} = s_{22}.$$

La propriété d'accord de clefs est alors violée pour une des deux sessions de la spécification donnée ci-dessus quand :

$$X_3 \neq X_6 \text{ ou } X_7 \neq s_{11}$$
ou
$$X_{13} \neq X_{10} \text{ ou } X_{14} \neq s_{22}.$$

Ainsi,

$$SC_\mathcal{P} = \{X_3 \neq X_6 \vee X_7 \neq s_{11} \vee X_{13} \neq X_{10} \vee X_{14} \neq s_{22}\}$$

4.6.3 Procédure de recherche d'attaques

Nous présentons dans cette section l'algorithme de recherche d'attaques proposé. Nous définissons cet algorithme comme suit :

Cet algorithme admet comme entrée deux paramètres : un protocole à tester et la(les) propriété(s) qu'il doit vérifier. Nous rappelons qu'un protocole est donné par $(\{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p\in\mathcal{P}}, <_\mathcal{P}, CI_0)$ avec $(\{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p\in\mathcal{P}}$ l'ensemble des pas du protocole, $<_\mathcal{P}$ est un ordre partiel sur l'ensemble de pas du protocole et CI_0 est l'ensemble des connaissances initiales de l'intrus. Quant

Algorithm 1 *Recherche-Attaques*(Pptes,Protocole)

```
ExecCorrect := Vrai
Instances := Construire-Ordonnancements-possibles(Protocole)
SC_P := Contraintes-Propriétés(Pptes,Protocole)
Tant que (ExecCorrect) Et (Instances ≠ ∅) Faire
  choisir (PT,PAT,CI,SC) ∈ Instances
  PeutComposer := Vrai
  Tant que (PeutComposer) Et PAT ≠ ∅ Faire
    choisir R_p → S_p ∈ PAT t.q p est minimal
    Si Composer(R_p,CI) Alors
      Traiter(R_p → S_p,CI,SC)
      Retirer-Ajouter(R_p → S_p, PAT, PT)
    Sinon
      PeutComposer := Faux
    FinSi
  Fin
  Si (PAT = ∅) Et (Attaque(SC,SC_P)) Alors
    ExecCorrect := Faux
  FinSi
 Fin
Fin Algorithme
```

à la propriété de sécurité, un exemple est de considérer l'authentification par rapport à un participant, ou le secret ou encore l'accord de clefs.

La première étape de la procédure de la recherche d'attaques consiste à générer l'ensemble de contraintes ($SC_{\mathcal{P}}$) relatives à la violation de la ou des propriété(s) à vérifier, donnée(s) en paramètre par *Pptes*. Cette génération est effectuée dans l'algorithme par la fonction *Contraintes-Propriétés*. Les étapes suivies pour la génération des contraintes décrivant la violation des propriétés sont :

- modélisation du protocole donné par le paramètre *Protocole* dans le modèle de services décrit dans le Chapitre 4 ;
- modélisation de la ou des propriété(s) donnée(s) par le paramètre *Pptes* dans le modèle de services ;
- déduction de l'ensemble de contraintes relatives à la violation de ces propriétés de sécurité à vérifier.

L'ensemble de contraintes généré constitue le premier niveau de l'arbre de contraintes qui sera expliquée dans la Section 4.6.4.

L'idée derrière cet algorithme est de considérer tous les ordonnancements possibles du protocole, qui sont définis par des différents ordres d'exécution $<_e$ sur $\mathcal{P}$, à la recherche d'une substitution close σ qui correspondrait (avec l'ordre d'exécution choisi) à une attaque. À chaque ordre d'exécution, nous associons une instance du protocole qui est définie par le quadruple (PT,PAT,CI,CS) avec,

- PT : l'ensemble de pas de l'exécution appartenant à l'ensemble $\{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p\in\mathcal{P}}$ qui ont été déjà traités. Ce sont des messages qui ont été déjà échangés entre les participants honnêtes et l'intrus. Au début de la procédure, cet ensemble est vide.
- PAT : l'ensemble de pas de l'exécution appartenant à $\{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p\in\mathcal{P}}$ qui ne sont pas

encore traités. Nous rappelons que cet ensemble est muni d'un ordre total qui permet de définir quel pas doit être exécuté avant un autre et par la suite quel message doit être échangé avant un autre. Initialement, cet ensemble contient tous les pas de l'exécution choisie dans $Instances$ dépendant du choix de l'ordre total sur $\mathcal{P}$.

- CI : l'ensemble des connaissances de l'intrus après avoir traité les pas de PT. Initialement, cet ensemble est CI_o donné par l'instance du protocole qui est en paramètre (*Protocole*).
- SC : l'ensemble de contraintes du protocole après avoir traité les pas de PT. Au début de la procédure, cet ensemble est vide puisqu'on n'a pas encore traité de pas. Cet ensemble de contraintes est représenté par l'arbre de contraintes décrit en détail dans la Section 4.6.4.

Ainsi, initialement, l'ensemble $Instances$ contient l'ensemble des instances de la forme $\{(\emptyset, \{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p\in\mathcal{P}}, CI_0, \emptyset)\}$ où $\{\mathcal{R}_p \rightarrow \mathcal{S}_p\}_{p\in\mathcal{P}}$ est un ensemble totalement ordonné de pas de protocole et CI_0 désigne l'ensemble de connaissances initiales de l'intrus. La fonction *Construire-Ordonnancements-possibles*(Protocole) génère tous les ordonnancements possibles en choisissant différents ordres d'exécution $<_e$ basés sur l'ordre $<_\mathcal{P}$ du protocole.

Ensuite, l'ensemble des variables utilisées dans l'algorithme est donné par la Table 4.3.

Considérons une instance parmi l'ensemble fini d'instances possibles ($Instances$). Une instance correspond à une attaque (existence d'une substitution close) si les contraintes (SC) générées tout au long des pas de cette instance sont cohérentes avec les contraintes déduites de la violation de la ou des propriété(s) à vérifier ($SC_\mathcal{P}$). Une contrainte est dite cohérente avec une autre contrainte si la conjonction de ces deux contraintes ne mène pas à $\perp$.

En effet, pour chaque pas de l'instance, nous considérons la règle qui lui correspond : $\mathcal{R}_p \rightarrow \mathcal{S}_p$. L'objectif de l'intrus est de composer un terme qui correspond en forme au terme $\mathcal{R}_p$ attendu par le participant à ce pas. Dans l'algorithme, ceci est testé par la fonction *Composer*. Cette fonction permet de tester si l'intrus peut composer le message $\mathcal{R}_p$ à partir de ses connaissances courantes. Le but de cette fonction est de générer les différentes contraintes qui correspondent aux différentes possibilités effectuées par l'intrus pour générer le message à composer. La fonction *Composer* retourne un résultat booléen. En effet, s'il existe au moins une possibilité pour composer le message alors le résultat est positif. Sinon, le résultat est négatif et dans ce cas, l'intrus a échoué à composer le message attendu par le participant et donc il ne peut pas continuer cette instance. Ainsi, l'instance considérée n'est pas exécutable et donc ne peut pas conduire à une attaque. Nous considérons alors une autre instance toujours à la recherche d'éventuelles attaques. Si le résultat est positif, alors nous traitons le pas courant de l'exécution $\mathcal{R}_p \rightarrow \mathcal{S}_p$ par la fonction *Traiter* en focalisant essentiellement sur deux points :

- l'ajout de l'ensemble de contraintes représentant la composition par l'intrus du message attendu dans le pas considéré ;
- la mise à jour des connaissances de l'intrus.

L'ensemble de contraintes relatives à un pas de l'exécution représente les différentes alternatives que l'intrus peut faire pour composer le message attendu dans ce pas. Par exemple, si ce message attendu t est de la forme $\{u\}_k^s$ alors l'intrus a deux possibilités : soit il obtient t par décomposition de ses connaissances, soit il l'obtient par composition et donc il a besoin de composer tout d'abord les deux termes u et k. Les contraintes que nous pouvons avoir dans notre algorithme sont, soit des contraintes d'égalité entre termes, soit d'inégalité (pour les propriétés), soit des contraintes $Composer(t, S)$ pour dire que le terme t doit être construit à partir de l'ensemble de termes E. Notons que, pour un seul pas de l'exécution, surtout quand l'intrus a plusieurs alternatives pour composer le message attendu, il peut y avoir un important nombre de contraintes à ajouter à l'ensemble de contraintes du protocole. D'où le besoin de gérer d'une manière optimale toutes ces contraintes. Cette gestion de contraintes sera expliquée en détail

Variable	**Fonctionalité**
Pptes	La ou les propriété(s) à vérifier.
Protocole	Il s'agit d'un triplet constitué de : $\mathcal{R}_p \rightarrow \mathcal{S}_p$: un ensemble de pas de protocole, $<_{\mathcal{P}}$: un ordre partiel sur cet ensemble de pas, CI_0 : l'ensemble de connaissances initiales de l'intrus.
ExecCorrect	Variable booléenne. Elle indique si le protocole est correct (vrai) ou s'il y a une attaque (faux).
Instances	L'ensemble d'instances possibles qui dépend essentiellement du choix de l'ordre d'exécution $<_e$, basé sur l'ordre partiel $<_{\mathcal{P}}$ du protocole. Chaque instance est un quadruplet constitué de : PT : les pas déjà traités. PAT : les pas à traiter. Cette ensemble est ordonné selon l'ordre total $<_e$. CI : les connaissances de l'intrus après avoir traité les pas de PT. SC : l'ensemble actuel de contraintes du protocole. Il est représenté par un arbre de contraintes (voir Section 4.6.4).
$SC_{\mathcal{P}}$	L'ensemble de contraintes relatives à la violation de la ou des propriétés de sécurité à vérifier. Cet ensemble forme le premier niveau de l'arbre de contraintes.
PeutComposer	Variable booléenne. Elle indique, pour une instance particulière de l'ensemble *Instances*, pour un pas bien déterminé $\{\mathcal{R}_p \Rightarrow \mathcal{S}_p\}$, si l'Intrus arrive à composer au moins un message qui correspond au message attendu ($\mathcal{R}_p$). Elle indique donc le cas où l'instance courante est exécutable ; sinon, il faut essayer une autre instance.

TAB. 4.3 – Récapitulatif des variables utilisées dans l'algorithme

dans la Section 4.6.4. Une fois le message composé, l'intrus acquiert de nouvelles connaissances qui seront utilisées dans les pas suivants. Ainsi, les connaissances de l'intrus CI doivent être mises à jour à chaque acquisition de nouvelles informations et par la suite à chaque traitement de pas de l'exécution considérée.

À la fin de l'instance, l'ensemble de contraintes constitué des deux ensembles de contraintes SC et $SC_{\mathcal{P}}$ est résolu afin de trouver une solution qui correspondrait à une violation de la ou des propriété(s) que le protocole était supposé vérifier. Cette résolution est assurée par la fonction *Attaque*. Cette fonction a pour objectif de tester si les deux ensembles de contraintes sont cohérents et donc s'il existe une solution pour leur résolution. Les deux ensembles sont cohérents si et seulement si, à la fin de l'exécution considérée, il existe au moins un chemin dans l'arbre de contraintes qui contient uniquement des contraintes cohérentes. Il est à noter que l'application des différentes optimisations et plus particulièrement, du test de cohérence et de l'élimination de la branche dans les cas de non cohérence permet de dire que les deux ensembles SC et $SC_{\mathcal{P}}$ sont cohérents s'il existe une branche complète dans l'arbre de contraintes.

Si les deux ensembles de contraintes sont cohérents alors la propriété testée n'est pas satisfaite pour l'exécution concernée. Dans ce cas, la fonction *Attaque* permet la résolution les contraintes des deux ensembles SC et $SC_{\mathcal{P}}$. L'instantiation des variables par la solution trouvée nous donne la trace de l'exécution correspondant à une attaque. Toutes les fonctions utilisées dans l'algorithme sont résumées dans la Table 4.4.

4.6.4 Gestion des contraintes

Notre méthode de recherche d'attaques est basée sur la résolution de contraintes. Le première partie des contraintes provient de la modélisation dans le modèle de services (voir Chapitre 4) de la violation de la ou des propriété(s) de sécurité à vérifier. La deuxième partie des contraintes est générée et mise à jour à chaque traitement d'un pas parmi l'ensemble de pas de l'instance considérée. Nous rappelons que cet ensemble de pas est muni d'un ordre total $<_e$ dans une telle instance. Vu que notre objectif est la falsification du protocole, notre procédure a pour rôle de trouver une instance qui correspond à deux ensembles de contraintes (SC et SC_P) cohérents.

Connaissant l'ensemble de contraintes SC_P, au cours du traitement d'un pas particulier d'une exécution, l'idée est d'ajouter uniquement les contraintes nécessaires, cohérentes avec les autres contraintes des pas précédents et ce afin de minimiser le nombre de contraintes ajoutées pour ce pas. Pour ce faire, nous proposons d'associer à l'arbre représentant l'exécution un arbre de contraintes. L'arbre de contraintes est initialement construit à partir des contraintes relatives à la violation de la ou des propriétés de sécurité que le protocole doit vérifier. Ces contraintes représentent le premier niveau de l'arbre de contraintes. Nous notons V l'ensemble de variables présentes dans l'ensemble courant des contraintes du protocole.

Cet ensemble contient initialement les variables existant au premier niveau de l'arbre de contraintes. Puis, comme les contraintes du premier niveau de l'arbre peuvent expliquer les différents choix pour violer la propriété de sécurité à vérifier, le niveau en question est composé de différents états représentant ces choix. Pour chacun de ces états, nous considérons les instances possibles (qui dépendent essentiellement du choix de l'ordre total) tout en cherchant une instance qui fournit un ensemble de contraintes cohérentes avec l'état en question et par la suite la substitution close qui est relative au chemin en question correspond à une attaque.

Ensuite, l'arbre doit être mis à jour pour chaque pas $\mathcal{R}_p \rightarrow \mathcal{S}_p$ à traiter. Supposons que

Fonction	**Paramètres**	**Rôle**
Contraintes-Propriétés	un protocole et la ou les propriétés à vérifier.	Cette fonction calcule le système de contraintes relatif à la ou les propriété(s) passée(s) en paramètre. Elle renvoie donc une disjonction de contraintes représentant les différentes possibilités pour violer ces propriétés.
Composer	le message à composer et l'ensemble de connaissances que doit utiliser l'intrus.	Cette fonction teste si l'Intrus peut composer le message attendu par le participant à partir de ses connaissances.
Traiter	un pas de protocole, l'ensemble de connaissances de l'intrus et l'ensemble de contraintes du protocole.	Cette fonction traite le pas donné en paramètre. Elle ajoute à l'ensemble de contraintes de protocole donné en paramètres les contraintes résultantes du traitement du pas considéré. Cette fonction a aussi pour rôle de mettre à jour les connaissances de l'intrus en leur ajoutant les nouvelles connaissances acquises pendant ce pas.
Retirer-Ajouter	un pas de protocole, l'ensemble des pas à traiter et l'ensemble de pas déjà traités.	Cette fonction enlève le pas donné en paramètre de l'ensemble de pas à traiter pour l'ajouter à l'ensemble des pas déjà traités.
Attaque	les deux ensembles de contraintes : celui de l'instance considérée et celui de la violation de la ou les propriétés à vérifier.	Cette fonction teste si les deux ensembles de contraintes sont cohérents. Si c'est le cas, la propriété de sécurité considérée est violée pour l'instance en question. La fonction *Attaque* résout alors la conjonction de ces deux ensembles de contraintes afin de trouver une solution. L'instanciation des variables par la solution trouvée dans l'exécution en cours donne la trace d'exécution de l'attaque.

TAB. 4.4 – Récapitulatif des fonctions utilisées dans l'algorithme

le niveau inférieur de l'arbre de contraintes est le niveau i. Pour chaque état du niveau i, nous considérons juste les contraintes qui découlent du traitement du pas courant et qui sont cohérentes avec les contraintes de l'état de niveau i en question. Une fois que l'intrus est capable de composer un message correspondant au message $\mathcal{R}_p$, nous générons donc plusieurs alternatives pour les contraintes permettant de générer $\mathcal{R}_p$ attendu par un participant honnête.

Vu que ces contraintes forment les différents choix pour composer le message attendu, elles représentent alors les différents états du niveau $i+1$. Soit EC_{i+1} l'ensemble de contraintes représentant un état du niveau $i+1$. La mise à jour de l'arbre de contraintes dépend de plusieurs cas :

- Les contraintes de EC_{i+1} ne contiennent pas de variables qui existaient déjà dans un état faisant partie du chemin liant l'état parent (appartenant au niveau i) à la racine. Dans ce cas, nous sauvegardons juste l'information que la variable en question (le message en général) doit être générée à partie d'un certain ensemble de connaissances (l'ensemble courant des connaissances de l'intrus). Cette information sera utile dans le cas où un niveau supérieur j ($j > i+1$) utilise cette même variable.
- Les contraintes de EC_{i+1} contiennent des variables qui existaient déjà dans un état faisant partie du chemin liant l'état parent (appartenant au niveau i) à la racine. Dans ce cas, il y a deux possibilités :
  - Les contraintes de EC_{i+1} sont incohérentes avec les contraintes des états parents. Dans ce cas, nous n'ajoutons pas d'états au niveau $(i+1)$. Dans le cas où nous n'ajoutons aucun état fils de niveau $(i+1)$ à l'état que nous sommes en train de traiter, nous supprimons la branche menant à cet état.
  - Les contraintes de EC_{i+1} sont cohérentes avec les contraintes des états parents Dans ce cas, nous maintenons les contraintes EC_{i+1} comme différents états fils de l'état père considéré appartenant au niveau i. Ensuite, si les contraintes d'un de ces états possibles contiennent une contrainte qui existe déjà dans un état parent ou pouvant être déduite par transition d'une autre contrainte précédente, nous n'ajoutons pas cette contrainte pour éviter la redondance.
    Puis, si une contrainte de EC_{i+1} gère des variables qui existent déjà dans V (ces variables ont donc des valeurs) en les reliant à des variables qui n'existent pas encore dans V, alors cette contrainte est remplacée par une nouvelle contrainte reliant la nouvelle variable à sa valeur (par transitivité).

4.7 Application de la méthode d'analyse

Nous présentons dans cette section les protocoles étudiés par notre méthode. Nous donnons tout d'abord un récapitulatif des résultats trouvés pour ces protocoles (Table 4.5).

Nous détaillons l'analyse du protocole Asokan Ginzboorg dans la Section 4.7.1. Nous donnons les résultats trouvés pour les deux autres protocoles : le protocole GDH.2 (Section 4.7.2) et le protocole (Section 4.7.3) sans détailler leurs analyses.

4.7.1 Protocole Asokan-Ginzboorg

Nous considérons la spécification du protocole de Asokan-Ginzboorg et la propriété de sécurité décrites dans le paragraphe 4.6.2. Nous considérons l'ordre d'exécution suivant ($<_e$) :

$$pas_{111} <_e pas_{212} <_e pas_{211} <_e pas_{112} <_e pas_{121} <_e pas_{222} <_e pas_{221} <_e pas_{122} <_e pas_{131} <_e pas_{232}.$$

Protocole étudié	**Exécution considérée**	**Résultat trouvé**
Asokan-Ginzboorg	une exécution de deux sessions en parallèle. Chaque session comprend deux participants.	Une attaque d'accord de clefs dans les deux sessions.
GDH.2	une exécution à quatre participants.	– une attaque d'authentification par rapport au participant P_3. – une attaque d'authentification par rapport au participant P_4.
	une exécution à n participants	– une attaque d'authentification par rapport à un participant intermédiaire. – une attaque d'authentification par rapport au dernier participant.
A-GDH.2	une exécution à n participants.	subdiviser le groupe en deux sous-groupes ayant deux vues différentes de clefs. L'intrus possède ces deux vues de clefs.

TAB. 4.5 – Synthèse des protocoles étudiés par la méthode de détection d'attaques

En appliquant notre méthode pour la spécification considérée du protocole, nous avons neuf pas de protocoles à traiter. Dans ce qui suit, nous donnons les paramètres de la fonction *Composer* pour chaque pas à traiter et l'ensemble de connaissances courantes de l'intrus.

$$pas_{111} \quad CS_1 = \{a_1, \{e_1\}_p\}$$

$$pas_{212} \quad CS_2 = CS_1 \cup \{a_2, \{e_2\}_p\}$$

$$pas_{211} \quad \left| \begin{array}{l} Composer(\langle X_4, \{X_5\}_p \rangle, CS_2) \\ CS_3 = CS_2 \cup \{a_2, \{r_2, s_{21}\}_{X_5}\} \end{array} \right.$$

$$pas_{112} \quad \left| \begin{array}{l} Composer(\langle X_{11}, \{X_{12}\}_p \rangle, CS_3) \\ CS_4 = CS_3 \cup \{a_1, \{r_1, s_{12}\}_{X_{12}}\} \end{array} \right.$$

$$pas_{121} \quad \left| \begin{array}{l} Composer(\langle X_1, \{X_2, X_3\}_{e_1} \rangle, CS_4) \\ CS_5 = CS_4 \cup \{\{X_3, s_{11}\}_{X_2}\} \end{array} \right.$$

$$pas_{222} \quad \left| \begin{array}{l} Composer(\langle X_8, \{X_9, X_{10}\}_{e_2} \rangle, CS_5) \\ CS_6 = CS_5 \cup \{\{X_{10}, s_{22}\}_{X_9}\} \end{array} \right.$$

$$pas_{221} \quad \left| \begin{array}{l} Composer(\{X_6, X_7\}_{r_2}, CS_6) \\ CS_7 = CS_6 \cup \{a_2, \{s_{21}, h(X_6, X_7)\}_{f(X_6, X_7)}\} \end{array} \right.$$

$$pas_{122} \left| \begin{array}{l} Composer(\{X_{13}, X_{14}\}_{r_1}, CS_7) \\ CS_8 = CS_7 \cup \{a_1, \{s_{12}, h(X_{13}, X_{14})\}_{f(X_{13},X_{14})}\} \end{array} \right.$$

$$pas_{131} \left| \begin{array}{l} Composer(\langle X_1, \{X_3, h(X_3, s_{11})\}_{f(X_3,s_{11})}\rangle, CS_7) \\ CS_9 = CS_8 \end{array} \right.$$

$$pas_{232} \left| \begin{array}{l} Composer(\langle X_8, \{X_{10}, h(X_{10}, s_{22})\}_{f(X_{10},s_{22})}\rangle, CS_9) \\ CS_{10} = CS_8 \end{array} \right.$$

Nous donnons dans la Table 4.6 les résultats trouvés (les formes normales des contraintes) par la fonction *Composer* pour chacun des pas du protocole Asokan.

pas_{211}	$(Composer(X_4, CS_2) \wedge X_5 = e_1) \vee (Composer(X_4, CS_2) \wedge X_5 = e_2)$
pas_{112}	$(Composer(X_{11}, CS_3) \wedge X_{12} = e_1) \vee (Composer(X_{11}, CS_3) \wedge X_{12} = e_2)$ $\vee (Composer(X_{11}, CS_3) \wedge X_{12} = \langle r_2, s_{21}\rangle \wedge X_5 = p)$
pas_{121}	$(Composer(X_1, CS_4) \wedge X_2 = r_1 \wedge X_3 = s_{12} \wedge X_{12} = e_1)$ $\vee (Composer(X_1, CS_4) \wedge X_2 = r_2 \wedge X_3 = s_{21} \wedge X_5 = e_1)$
pas_{222}	$(Composer(X_8, CS_5) \wedge X_9 = X_3 \wedge X_{10} = s_{11} \wedge X_2 = e_2)$ $\vee (Composer(X_8, CS_5) \wedge X_9 = r_1 \wedge X_{10} = s_{12} \wedge X_{12} = e_2)$ $\vee (Composer(X_8, CS_5) \wedge X_9 = r_2 \wedge X_{10} = s_{21} \wedge X_5 = e_2)$
pas_{221}	$(X_6 = X_{10} \wedge X_7 = s_{22} \wedge X_9 = r_2) \vee (X_6 = X_3 \wedge X_7 = s_{11} \wedge X_2 = r_2)$ $\vee (X_6 = r_1 \wedge X_7 = s_{12} \wedge X_{12} = r_2) \vee (X_6 = r_2 \wedge X_7 = s_{21} \wedge X_5 = r_2)$ $\vee (Composer(X_6, CS_6) \wedge Composer(X_7, CS_6) \wedge Composer(X_9, CS_6) \wedge X_{10} = r_2)$ $\vee (Composer(X_6, CS_6) \wedge Composer(X_7, CS_6) \wedge Composer(X_2, CS_6) \wedge X_3 = r_2)$ $\vee (Composer(X_6, CS_6) \wedge Composer(X_7, CS_6) \wedge Composer(X_5, CS_6))$
pas_{122}	$(X_{13} = X_{10} \wedge X_{14} = s_{22} \wedge X_9 = r_1) \vee (X_{13} = X_3 \wedge X_{14} = s_{11} \wedge X_2 = r_1)$ $\vee (X_{13} = r_1 \wedge X_{14} = s_{12} \wedge X_{12} = r_1) \vee (X_{13} = r_2 \wedge X_{14} = s_{21} \wedge X_5 = r_1)$ $\vee (Composer(X_{13}, CS_7) \wedge Composer(X_{14}, CS_7) \wedge Composer(X_9, CS_7) \wedge X_{10} = r_1)$ $\vee (Composer(X_{13}, CS_7) \wedge Composer(X_{14}, CS_7) \wedge Composer(X_2, CS_7) \wedge X_3 = r_1)$ $\vee (Composer(X_{13}, CS_7) \wedge Composer(X_{14}, CS_7) \wedge Composer(X_{12}, CS_7))$
pas_{131}	$(Composer(X_1, CS_8) \wedge X_3 = s_{21} \wedge X_7 = s_{11} \wedge X_6 = X_3)$ $\vee (Composer(X_1, CS_8) \wedge X_3 = s_{12} \wedge X_{14} = s_{11} \wedge X_{13} = X_3)$
pas_{232}	$(Composer(X_8, CS_8) \wedge X_{10} = s_{12} \wedge X_{14} = s_{22} \wedge X_{13} = X_{10})$ $\vee (Composer(X_8, CS_8) \wedge X_{10} = s_{21} \wedge X_7 = s_{22} \wedge X_6 = X_{10})$

TAB. 4.6 – Contraintes en formes normales du protocole Asokan Ginzboorg

Par application de notre méthode d'analyse pour l'ordre d'exécution considéré, nous avons obtenu une branche d'un arbre de contraintes illustrée par la Figure 4.7 :

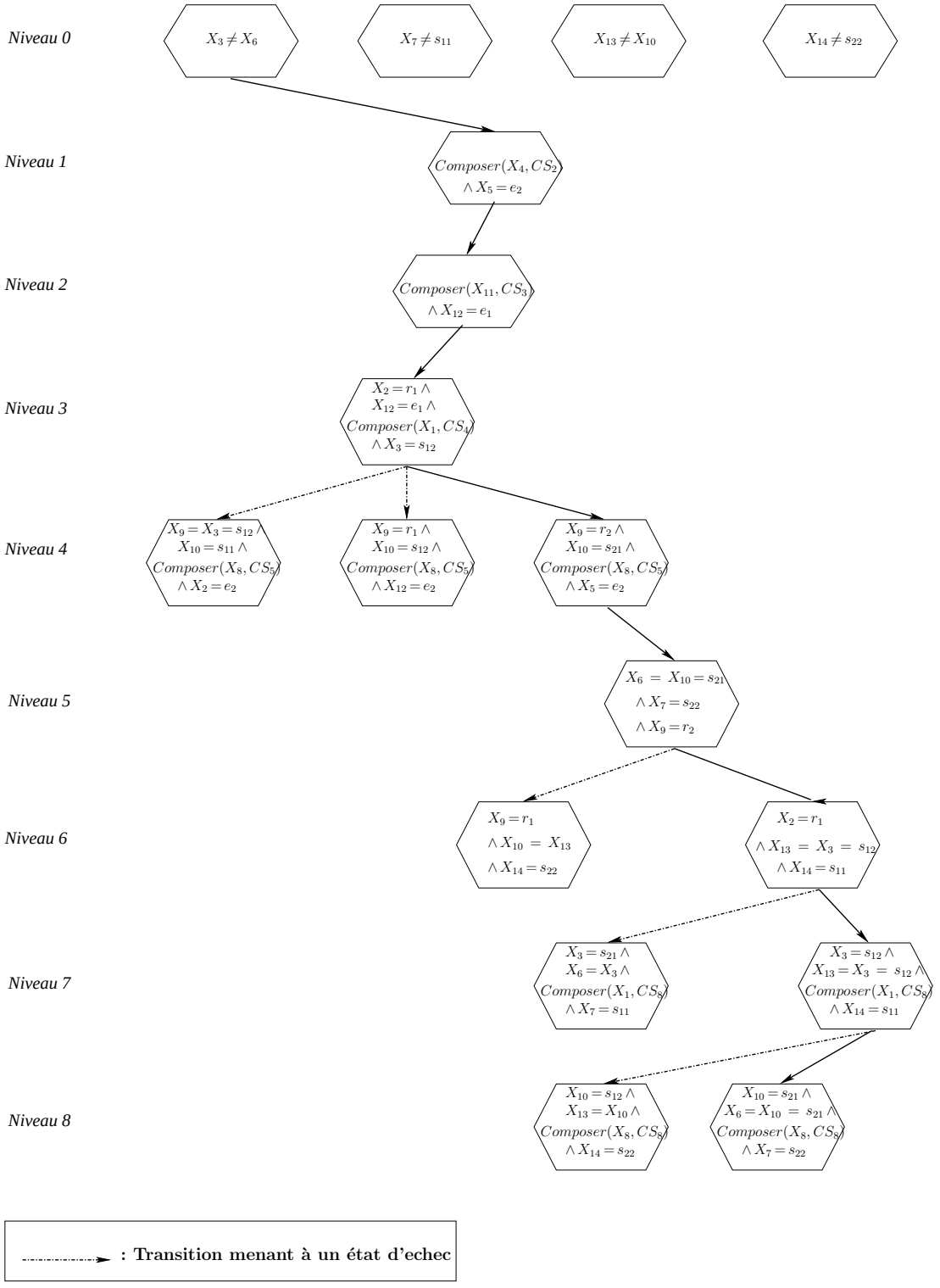


FIG. 4.7 – Arbre de contraintes pour le protocole Asokan-Ginzboorg

La solution σ qui correspondrait à une attaque est donc donnée par :

$$\begin{array}{lcl|lcl} X_2\sigma & = & r_1 & X_3\sigma & = & s_{12} \\ X_5\sigma & = & e_2 & X_6\sigma & = & s_{21} \\ X_7\sigma & = & s_{22} & X_9\sigma & = & r_2 \\ X_{10}\sigma & = & s_{21} & X_{10}\sigma & = & s_{21} \\ X_{12}\sigma & = & e_1 & X_{14}\sigma & = & s_{11} \end{array}$$

En considérant σ définie ci-dessus, nous obtenons l'exécution décrite par la Figure 4.8.

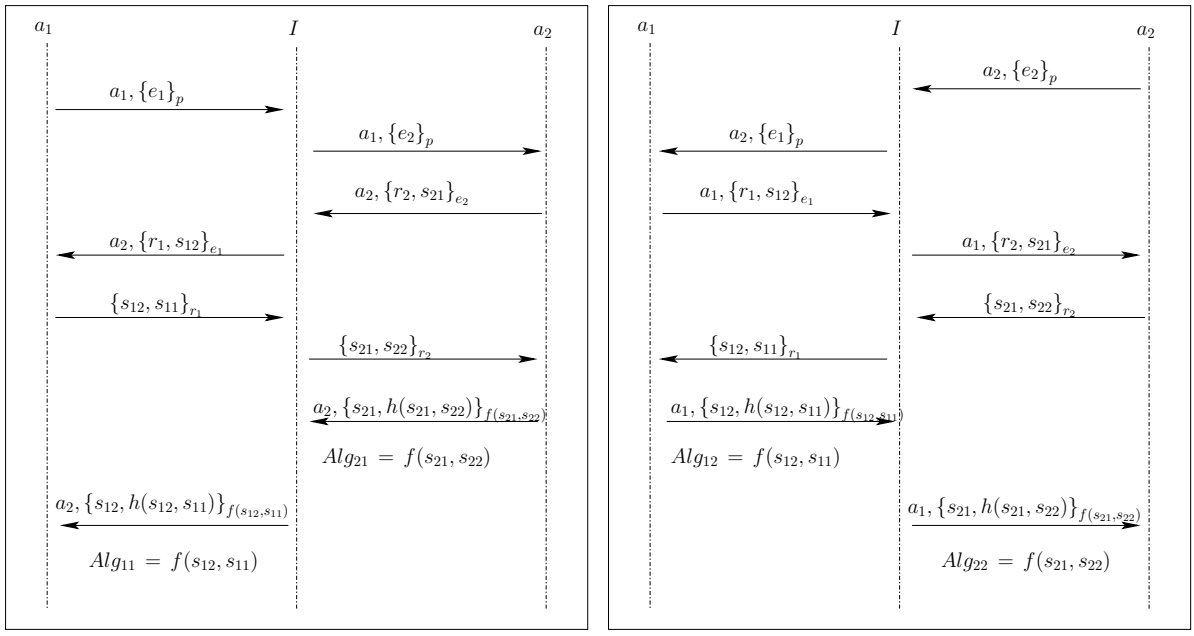


FIG. 4.8 – Attaque d'accord de clefs sur le protocole Asokan-Ginzboorg

À la fin de cette exécution, nous avons bien :

$$\begin{array}{rcl} & Alg_{11} & \neq & Alg_{21} \\ et & Alg_{12} & \neq & Alg_{22} \end{array}$$

Ainsi la propriété d'Accord de clefs n'est pas vérifiée pour chacune des deux sessions.

4.7.2 Protocole GDH.2

Le protocole GDH.2 [8] met en communication n participants. L'objectif est de calculer une clef commune à tous les membres appelée clef du groupe. Cette clef est calculée à partir des contributions des membres. La liste des membres est supposée ordonnée. En recevant un message (concaténation d'exponentielles) de son précédent voisin, un membre va exponentier les composantes du message reçu par une information privée et envoyer le nouveau message au voisin suivant. Ce traitement est valable pour tous les membres mis à part le premier et le dernier membre. Le premier participant déclenche la génération de la clef. Il va envoyer sa contribution (en exponentielle) au voisin suivant. Quant au dernier membre, en recevant le message attendu (composantes exponentielles), il va considérer la dernière composante pour calculer sa clef de groupe et va exponentier les autres composantes par une information privée pour les diffuser au reste du groupe.

Analyse du protocole GDH.2

Nous présentons dans cette section l'analyse effectuée pour le protocole GDH.2. Nous présentons tout d'abord la session considérée de ce protocole ainsi que la modélisation de cette session et

celle de la propriété à vérifier. Nous présentons ensuite le résultat obtenu pour cette session et pour cette propriété.

Entrée de l'analyse du protocole GDH.2

Nous utilisons les mêmes notations définies dans le paragraphe 4.6.2. Nous considérons une session du protocole GDH.2 avec quatre participants : P_1, P_2, P_3 et P_4. Le participant P_1 joue le rôle de l'initiateur, P_2 et P_3 sont des rôles intermédiaires et P_4 joue le rôle du dernier membre du groupe. Cette session est illustrée par la Figure 4.9.

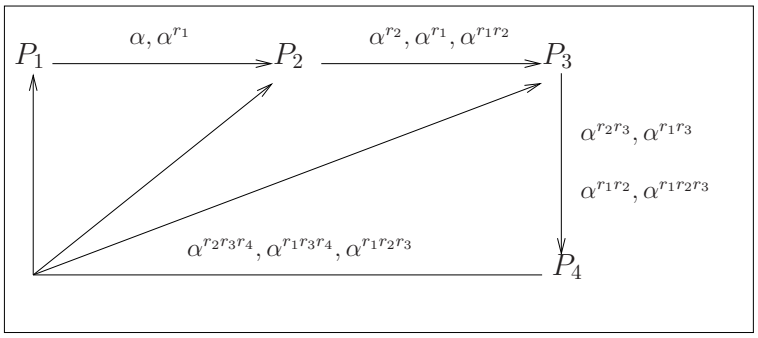


FIG. 4.9 – Le protocole GDH.2 avec 4 participants

Puisque nous considérons une seule session et pour plus de lisibilité, nous notons pas_{ij1} par pas_{ij} pour désigner le pas j du participant i. Quant à la vue de la clef du groupe par un participant i, elle est notée par Alg_i. Les deux messages fixes $Init$ et End permettent respectivement d'initier et de conclure une session. Les différents pas de protocole spécifiant les actions des participants P_1, P_2, P_3 et P_4 sont décrits comme suit :

$\mathbf{pas_{11}}$:	$Init$	$\longrightarrow$	$exp(\alpha, 1), exp(\alpha, r_1)$
$\mathbf{pas_{12}}$:	$exp(X_1, X_2), exp(X_3, X_4), exp(X_5, X_6)$	$\longrightarrow$	End $Alg_1 = exp(exp(X_1, X_2), r_1)$
$\mathbf{pas_{21}}$:	$exp(\alpha, 1), exp(\alpha, X_7)$	$\longrightarrow$	$exp(\alpha, r_2), exp(\alpha, X_7),$ $exp(exp(\alpha, X_7), r_2)$
$\mathbf{pas_{22}}$:	$exp(X_8, X_9), exp(X_{10}, X_{11}), exp(X_{12}, X_{13})$	$\longrightarrow$	End $Alg_2 = exp(exp(X_{10}, X_{11}), r_2)$
$\mathbf{pas_{31}}$:	$exp(\alpha, X_{14}), exp(\alpha, X_{15}), exp(X_{16}, X_{17})$	$\longrightarrow$	$exp(exp(\alpha, X_{14}), r_3),$ $exp(exp(\alpha, X_{15}), r_3),$ $exp(X_{16}, X_{17}),$ $exp(exp(X_{16}, X_{17}), r_3)$
$\mathbf{pas_{32}}$:	$exp(X_{18}, X_{19}), exp(X_{20}, X_{21}), exp(X_{22}, X_{23})$	$\longrightarrow$	End $Alg_3 = exp(exp(X_{22}, X_{23}), r_3)$

$\mathbf{pas_{41}}$:	$exp(X_{24}, X_{25}), exp(X_{26}, X_{27}), exp(X_{28}, X_{29}),$ $exp(X_{30}, X_{31})$	$\longrightarrow$	$exp(exp(X_{24}, X_{25}), r_4),$ $exp(exp(X_{26}, X_{27}), r_4),$ $exp(exp(X_{28}, X_{29}), r_4)$ $Alg_4 = exp(exp(X_{30}, X_{31}), r_4)$

L'ensemble $\mathcal{P} = \{pas_{11}, pas_{12}, pas_{21}, pas_{22}, pas_{31}, pas_{32}, pas_{41}\}$ est muni d'un ordre partiel $<_{\mathcal{P}}$ défini par : $pas_{11} <_{\mathcal{P}} pas_{12}$, $pas_{21} <_{\mathcal{P}} pas_{22}$ et $pas_{31} <_{\mathcal{P}} pas_{32}$.

La modélisation de cette session dans le modèle de services donne le résultat suivant :

$\mathbf{P_i}$	$\mathbf{S_i}$	$\mathbf{S_{c_i}}$	$\mathbf{K_i}$	$\cup_\mathbf{j}\mathbf{K_{ij}}$
P_1	$exp(X_1, X_2)$	$exp(\alpha, r_1)$	r_1	α
P_2	$exp(X_{10}, X_{11})$	$exp(\alpha, r_2)$, $exp(exp(\alpha, X_7), r_2)$	r_2	α
P_3	$exp(X_{22}, X_{23})$	$exp(exp(\alpha, X_{14}), r_3)$, $exp(exp(\alpha, X_{15}), r_3)$ $exp(exp(X_{16}, X_{17}), r_3)$	r_3	α
P_4	$exp(X_{30}, X_{31})$	$exp(exp(X_{24}, X_{25}), r_4)$, $exp(exp(X_{26}, X_{27}), r_4)$ $exp(exp(X_{28}, X_{29}), r_4)$	r_4	α

La propriété que nous voulons vérifier est le secret de la clef du groupe. Cette propriété est spécifiée dans le modèle de services décrit en Section 4.3 comme suit :

$$\forall P_i \in P, K_I \cup_j K_{Ij} \cup_{P_k \in P} S_{c_k} \not\models \mathcal{A}lg_i(K_i, \cup_j K_{ij}, S_i) \ .$$

La propriété de secret de groupe est violée quand l'intrus obtient au moins une vue d'un membre de groupe. L'acquisition d'une vue d'un participant est définie comme étant la violation de la propriété d'authentification par rapport au participant considéré.

Nous vérifions dans un premier temps l'authentification par rapport au participant P_4. Cette propriété est violée pour notre cas quand l'intrus obtient dans ses connaissances l'information $exp(exp(X_{30}, X_{31}), r_4)$. À partir de la modélisation en modèle de services décrite ci-dessus (et surtout des S_{c_i}), la violation de la propriété d'authentification par rapport au participant P_4 se traduit alors par le système de contraintes **SC2** suivant :

$$\begin{array}{lcl} X_{24} = X_{30} & et & X_{25} = X_{31} \\ & ou & \\ X_{26} = X_{30} & et & X_{27} = X_{31} \\ & ou & \\ X_{28} = X_{30} & et & X_{29} = X_{31} \end{array}$$

Ensuite, nous vérifions aussi l'authentification par rapport au participant P_3. D'une manière similaire, la violation de cette propriété se traduit par le système de contraintes **SC3** :

$$\begin{array}{lcl} X_{22} = \alpha & et & X_{23} = x_{14} \\ & ou & \\ X_{22} = \alpha & et & X_{15} = X_{23} \\ & ou & \\ X_{22} = X_{16} & et & X_{17} = X_{23} \end{array}$$

Résultat de l'analyse du protocole GDH.2

Nous considérons l'ensemble des pas de protocoles présenté dans le paragraphe précédent muni de l'ordre d'exécution total suivant $<_e$:
$pas_{11} <_e pas_{21} <_e pas_{31} <_e pas_{41} <_e pas_{12} <_e pas_{22} <_e pas_{32}$
L'intrus doit composer les messages attendus par les différents participants à partir de ses connaissances et ceci selon l'ordre d'exécution traité.
Selon les informations acquises, l'intrus essaye de composer le message attendu pour le pas courant. Un ensemble de contraintes liant les variables de ce message aux informations déjà acquises est alors construit. Si nous considérons l'exécution relative à l'ordre d'exécution défini auparavant, nous obtenons le système de contraintes **SC1** de la Table 4.7.

Une fois le système de contraintes du protocole établi, nous considérons maintenant les propriétés à vérifier. Prenons par exemple le cas de la première propriété à vérifier : l'authentification par rapport à P_4. La violation de cette propriété est caractérisée par le système de contrainte **SC2**. La résolution des deux systèmes de contrainte $SC1$ et $SC2$ permet de construire une attaque de la propriété testée. Nous remarquons ici que les deux systèmes de contraintes ne sont pas contradictoires. D'une manière similaire, nous obtenons aussi une attaque d'authentification par rapport à P_3 qui est définie par le système de contraintes SC_3.

Attaque d'authentification par rapport à P_4

Nous supposons dans cette exécution que $X_{24} = X_{30}$ et $X_{25} = X_{31}$. Nous obtenons alors :

pas_{21}	$((X_7 = 1) \vee (X_7 = r_1))$
pas_{31}	$((X_{14} = 1) \vee (X_{14} = r_1) \vee (X_{14} = r_2) \vee (X_{14} = X_7))$ $\wedge((X_{15} = 1) \vee (X_{15} = r_1) \vee (X_{15} = r_2) \vee (X_{15} = X_7))$ $\wedge$ $(((X_{16} = \alpha) \wedge (X_{17} = 1)) \vee ((X_{16} = \alpha) \wedge (X_{17} = X_7))$ $\vee((X_{16} = \alpha) \wedge (X_{17} = r_1)) \vee ((X_{16} = \alpha) \wedge (X_{17} = r_2))$ $((X_{16} = exp(\alpha, X_7)) \wedge (X_{17} = r_2)))$
pas_{41}	$((X_{24} = \alpha) \wedge (X_{25} = 1)) \vee ((X_{24} = \alpha) \wedge (X_{25} = X_7))$ $\vee((X_{24} = \alpha) \wedge (X_{25} = r_1)) \vee ((X_{24} = \alpha) \wedge (X_{25} = r_2))$ $\vee((X_{24} = exp(\alpha, X_7)) \wedge (X_{25} = r_2)) \vee ((X_{24} = exp(\alpha, X_{15})) \wedge (X_{25} = r_3))$ $\vee((X_{24} = X_{16}) \wedge (X_{25} = X_{17})) \vee ((X_{24} = exp(X_{16}, X_{17})) \wedge (X_{25} = r_3))$ *les couples* (X_{26}, X_{27}), (X_{28}, X_{29}) *et* (X_{30}, X_{31}) *ont les* mêmes *contraintes que le couple* (X_{24}, X_{25}) $(ci - dessus)$.
pas_{12}	$((X_1 = \alpha) \wedge (X_2 = 1)) \vee ((X_1 = \alpha) \wedge (X_2 = X_7))$ $\vee((X_1 = \alpha) \wedge (X_2 = r_1)) \vee ((X_1 = \alpha) \wedge (X_2 = r_2))$ $\vee((X_1 = exp(\alpha, X_7)) \wedge (X_2 = r_2)) \vee ((X_1 = exp(\alpha, X_{15})) \wedge (X_2 = r_3))$ $\vee((X_1 = X_{16}) \wedge (X_2 = X_{17})) \vee ((X_1 = exp(X_{16}, X_{17})) \wedge (X_2 = r_3))$ $\vee((X_1 = exp(X_{24}, X_{25})) \wedge (X_2 = r_4)) \vee ((X_1 = exp(X_{26}, X_{27})) \wedge (X_2 = r_4))$ $\vee((X_1 = exp(X_{28}, X_{29})) \wedge (X_2 = r_4))$
pas_{22}	*les couples* (X_8, X_9), (X_{10}, X_{11}) *et* (X_{12}, X_{13}) *ont les* mêmes *contraintes que le couple* (X_1, X_2) *(voir* pas_{12}*).*
pas_{32}	*les couples* (X_{18}, X_{19}), (X_{20}, X_{21}) *et* (X_{22}, X_{23}) *ont les* mêmes *contraintes que le couple* (X_1, X_2) *(voir* pas_{12}*).*

TAB. 4.7 – Contraintes en formes normales du protocole GDH.2

$$
\begin{array}{lrcll}
\mathbf{pas_{11}}: & I & \longrightarrow & P_1: & Init \\
 & P_1 & \longrightarrow & I: & exp(\alpha,1), exp(\alpha,r_1) \\
 & & & & \\
\mathbf{pas_{21}}: & I & \longrightarrow & P_2: & exp(\alpha,1), exp(\alpha,X_7) \\
 & P_2 & \longrightarrow & I: & exp(\alpha,r_2), exp(\alpha,X_7), exp(exp(\alpha,X_7),r_2) \\
 & & & & \\
\mathbf{pas_{31}}: & I & \longrightarrow & P_3: & exp(\alpha,X_{14}), exp(\alpha,X_{15}), exp(X_{16},X_{17}) \\
 & P_3 & \longrightarrow & I: & exp(exp(\alpha,X_{14}),r_3), exp(exp(\alpha,X_{15}),r_3), \\
 & & & & exp(X_{16},X_{17}), exp(exp(X_{16},X_{17}),r_3) \\
 & & & & \\
\mathbf{pas_{41}}: & I & \longrightarrow & P_4: & exp(X_{24},X_{25}), exp(X_{26},X_{27}), \\
 & & & & exp(X_{28},X_{29}), exp(X_{24},X_{25}) \\
 & P_4 & \longrightarrow & I: & exp(exp(X_{24},X_{25}),r_4), exp(exp(X_{26},X_{27}),r_4), \\
 & & & & exp(exp(X_{28},X_{29}),r_4) \\
 & & & & \mathbf{Alg_4 = exp(exp(X_{24}, X_{25}), r_4)}
\end{array}
$$

D'après cette exécution, à ce stade, l'intrus connaît $exp(exp(X_{24}, X_{25}), r_4)$ puisqu'il l'a acquis du participant P_4 pendant le pas pas_{41}. Néanmoins, cette information est la vue Alg_4 de la clef du participant P_4. La Figure 4.10 donne une instanciation de cette attaque.

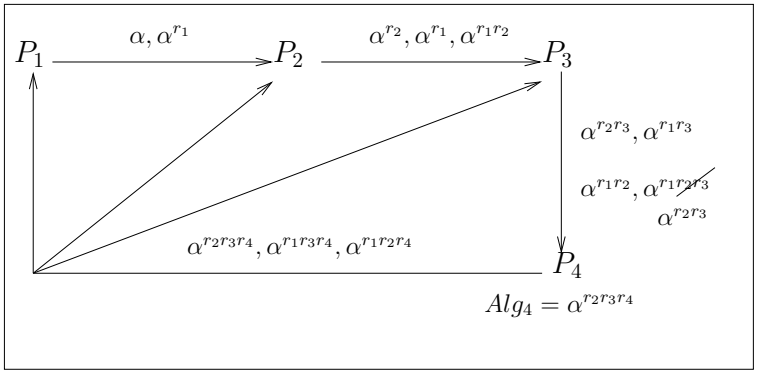


FIG. 4.10 – Première attaque d'authentification sur GDH.2 avec 4 participants

Attaque d'authentification par rapport à P_3
Nous supposons dans cette exécution par exemple que $X_{22} = \alpha$ et $X_{23} = X_{14}$.

$$
\begin{array}{lrcll}
\mathbf{pas_{11}}: & I & \longrightarrow & P_1: & Init \\
 & P_1 & \longrightarrow & I & exp(\alpha, 1), exp(\alpha, r_1) \\
 & & & & \\
\mathbf{pas_{21}}: & I & \longrightarrow & P_2: & exp(\alpha, 1), exp(\alpha, X_7) \\
 & P_2 & \longrightarrow & I & exp(\alpha, r_2), exp(\alpha, X_7), exp(exp(\alpha, X_7), r_2) \\
 & & & & \\
\mathbf{pas_{31}}: & I & \longrightarrow & P_3: & exp(\alpha, X_{14}), exp(\alpha, X_{15}), exp(X_{16}, X_{17}) \\
 & P_3 & \longrightarrow & I & exp(exp(\alpha, X_{14}), r_3), exp(exp(\alpha, X_{15}), r_3), \\
 & & & & exp(X_{16}, X_{17}), exp(exp(X_{16}, X_{17}), r_3) \\
 & & & & \\
\mathbf{pas_{41}}: & I & \longrightarrow & P_4: & exp(X_{24}, X_{25}), exp(X_{26}, X_{27}), \\
 & & & & exp(X_{28}, X_{29}), exp(X_{30}, X_{31}) \\
 & P_4 & \longrightarrow & I & exp(exp(X_{24}, X_{25}), r_4), exp(exp(X_{26}, X_{27}), r_4), \\
 & & & & exp(exp(X_{28}, X_{29}), r_4) \\
 & & & & \mathbf{Alg_4 = exp(exp(X_{30}, X_{31}), r_4)} \\
 & & & & \\
\mathbf{pas_{32}}: & I & \longrightarrow & P_3: & exp(X_{18}, X_{19}), exp(X_{20}, X_{21}), exp(\alpha, X_{14}) \\
 & P_1 & \longrightarrow & I & \mathbf{Alg_3 = exp(exp(\alpha, X_{14}), r_3)}
\end{array}
$$

D'après cette exécution, à ce stade, l'intrus connaît $exp(exp(\alpha, X_{14}), r_3)$ puisqu'il l'a acquis du participant P_3 pendant le pas pas_{32}. Néanmoins, cette information est la vue de la clef du participant P_3 Alg_3. La Figure 4.11 donne une instanciation de cette attaque.

Généralisation de l'attaque sur GDH.2

Nous avons vu que, pour le protocole GDH.2 avec quatre participants, l'intrus peut avoir la vue de la clef du groupe du participant P_4 ou celle du participant P_3. Les deux cas traitent la

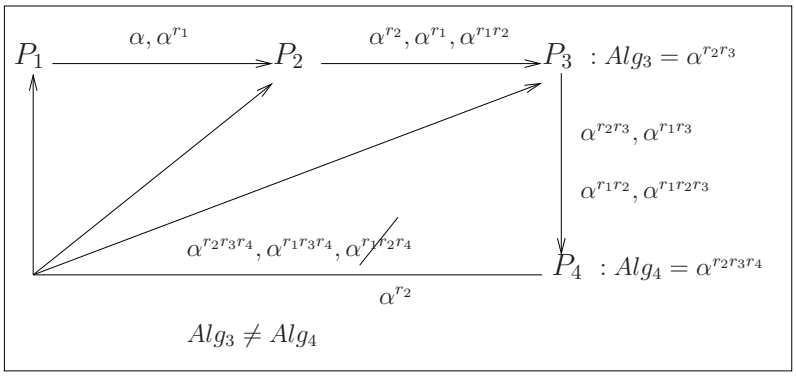


FIG. 4.11 – Deuxième Attaque d'authentification sur GDH.2 avec 4 participants

propriété d'authentification pour le protocole GDH.2 avec quatre participants. Ce résultat reste encore vrai si nous considérons le même protocole avec n participants. En effet, l'intrus peut avoir la vue de la clef de groupe du dernier membre ou celle d'un autre rôle intermédiaire (par exemple ici P_3).

Avoir la vue de la clef du dernier participant

Pour avoir la vue du dernier participant, l'intrus intercepte le dernier message destiné au dernier participant $(X_1, .., X_{n-1}, X_n)$ et modifie la dernière composante (X_n) en la remplaçant par n'importe quelle autre composante du message variant de X_1 à X_{n-1}. L'action du dernier participant est :

1. d'exponentier les composantes $X_1 ... X_{n-1}$ par r_n et transmettre (sur le réseau) cela aux autres participants. l'intrus peut donc avoir ces informations : $exp(X_1, r_n)$ à $exp(X_{n-1}, r_n)$.
2. d'utiliser la dernière composante X_n pour générer la clef du groupe en l'exponentiant par r_n. La vue de la clef du groupe pour le dernier participant P_n est alors $Alg_n = exp(X_n, r_n)$.

Ainsi, si l'intrus remplace X_n par un message X_i de X_1 à X_{n-1}, le dernier participant génère comme clef de groupe $Exp(X_i, r_n)$. Or, cette information est déjà disponible sur le réseau et donc accessible à l'intrus.

Avoir la vue de la clef d'un participant intermédiaire

L'objectif de l'intrus est maintenant d'avoir la vue de la clef d'un membre intermédiaire P_i variant de P_2 à P_{n-1}. Le participant P_i génère sa clef de groupe à partir du dernier message reçu X du dernier participant P_n : $X = X_1, .., X_{n-1}$. En plus, P_i ne va pas répondre à ce message en transmettant des messages sur le réseau. Or, la vue de la clef du groupe contient une information privée du participant P_i : r_i.

Ainsi, la seule possibilité pour l'intrus d'avoir la vue de P_i est de faire en sorte que P_i génère pour clef de groupe un message qui a été transmis avant et qui contient l'information privée r_i. Ces messages sont accessibles à l'intrus lors du premier tour du protocole : quand les membres (en particulier, les intermédiaires) sont invités à donner leurs contributions en exponentiant les messages reçus par les r_i (par exemple, pour GDH.2 à 4 participants, pour P_3, le pas pas_{31}).

Pour un participant intermédiaire P_i, le premier pas lors duquel, il va envoyer sa contribution est pas_i. P_i reçoit un message de la forme $X_{11}, .., X_{1i}$. Dans le message à envoyer par P_i pour ce pas, nous trouvons toutes les composantes X_{1j} (variant de X_{11} à X_{1i}) exponentiées par r_i. Nous

supposons maintenant que, pour un $i \in \{1, .., n-1\}$, pour le message $X = X_1, .., X_i, .., X_{n-1}$, X_i fait partie des composantes $X_{11}, .., X_{1i}$.

Le participant P_i, en recevant X, prend la composante qui l'intéresse : X_i et génère sa vue de la clef du groupe en l'exponentiant par r_i. Ainsi, $Alg_i = exp(exp(X_i, r_i))$. Or, l'intrus possède l'information $exp(X_i, r_i)$ et ceci depuis le pas pas_i. Ainsi, l'intrus peut avoir la vue d'un participant intermédiaire pour le protocole GDH.2 avec n participants.

4.7.3 Protocole A-GDH.2

Le protocole A-GDH.2 [8] comme le protocole GDH.2 met en communication n participants afin de calculer une clef commune à tous les membres appelée clef du groupe. Il doit assurer aussi l'authentification de la clef de groupe. Le fonctionnement est similaire à celui du protocole GDH.2 (Voir Section 4.7.2) à la différence que, pour le dernier participant, les messages sont exponentiés par son contribution à la clef du groupe (comme le protocole GDH.2) et les clefs partagées avec les autres participants.

Analyse du protocole A-GDH.2.

La violation de la propriété traitée dans le paragraphe 4.7.2 permet à l'intrus d'avoir la vue de la clef du groupe du participant attaqué. Cette vue n'est pas nécessairement la vue de la clef du groupe de tous les membres. Ainsi, l'intrus peut communiquer avec le membre attaqué ou se faire passer pour lui. Cet intrus est généralement un attaquant de l'extérieur.

Pour le protocole A-GDH.2 (voir Figure 4.12 pour une session particulière de ce protocole), l'intrus ne peut pas générer une attaque de la manière décrite dans la Section 4.7.2 puisque le dernier participant P_n exponentie les composantes, non plus par r_n seulement mais aussi par une clef partagée entre lui et le membre concerné.

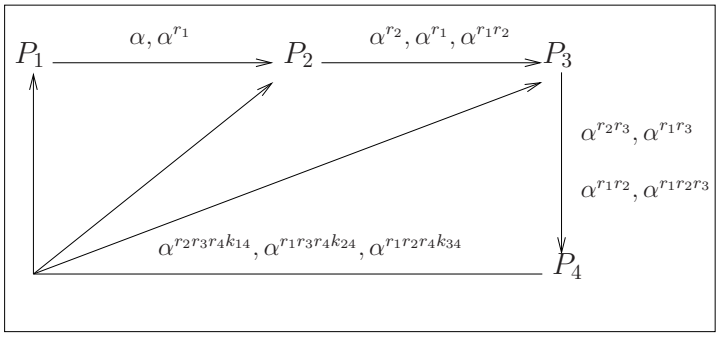


FIG. 4.12 – Le protocole A-GDH.2 avec 4 participants

Ainsi, l'intrus est contraint d'être à l'intérieur du groupe pour pouvoir mener une attaque. Son objectif n'est plus alors d'avoir la vue d'un participant puisqu'il fait partie du groupe (il a le droit donc d'avoir la clef du groupe). Néanmoins, il peut nuire au groupe en faisant en sorte que les membres du groupe aient chacun différente de la clef. L'objectif de l'intrus est alors de subdiviser le groupe en sous groupes de telle sorte que lui seul puisse communiquer (ait la même vue de la clef) avec tous les membres sans que les membres partagent la même clef.
Nous nous inspirons de l'attaque trouvée pour GDH.2 à 4 participants. Nous considérons donc le protocole A-GDH.2 à quatre participants où l'intrus joue le rôle du troisième participant.

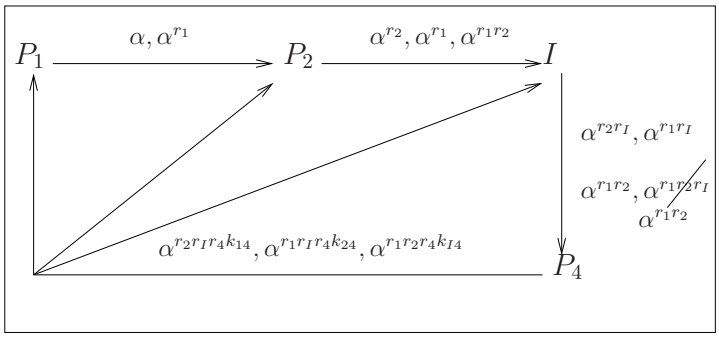


FIG. 4.13 – Attaque sur A-GDH.2 avec 4 participants

L'intrus se focalise sur le dernier participant P_4. Il modifie la dernière composante du message attendu par P_4 en la remplaçant par l'avant dernière composante. P_4 exponentie les trois premières composantes par r_4 et la clef partagée avec les participants concernés. Il utilise la dernière composante pour générer sa vue de clef en l'exponentiant par r_4. À la fin de cette exécution, les vues des participants (y compris l'intrus) sont :

- Vue de P_1 : $\alpha^{r_1 r_2 r_I r_4}$.
- Vue de P_2 : $\alpha^{r_1 r_2 r_I r_4}$.
- Vue de P_4 : $\alpha^{r_1 r_2 r_4}$.
- Vues de I : $\alpha^{r_1 r_2 r_I r_4}$ et $\alpha^{r_1 r_2 r_4}$.

L'intrus réussit donc à diviser le groupe en deux parties : la première partie comprend seulement le participant P_4 et la deuxième comprend le reste du groupe. En plus, l'intrus a les deux vues. Ainsi, il contrôle toutes les communications.

Généralisation de l'attaque sur A-GDH.2.

Nous considérons le protocole A-GDH.2 avec n participants : $P_1, .., P_n$. L'intrus fait partie du groupe et il a pour rang i : $I = P_i$. L'intrus se focalise sur le dernier message destiné au dernier participant. Ce message X comprend n composantes $X = X_1..X_n$. Le dernier participant va exponentier les $n-1$ composantes de ce message par r_n et par la clef du participant correspondant. Ainsi, la composante X_i va être exponentiée par $r_n k_{ni}$. Au lieu d'envoyer le message $X = X_1, .., X_i, .., X_n$, l'intrus envoie le message $X = X_1, .., X_i, .., X_i$ à P_n. P_n envoie comme réponse le message $X' = exp(exp(X_1, r_n), k_{n1}), .., exp(exp(X_i, r_n), k_{ni}), .., exp(exp(X_{n-1}, r_n), k_{n(n-1)})$ et génère comme clef de groupe $exp(X_n, r_n)$. À partir du message X', l'intrus a la composante $exp(exp(X_i, r_n), k_{ni})$. Il peut donc récupérer $exp(X_i, r_n) = exp(X_n, r_n)$. Ainsi, il possède la vue de la clef du participant P_n. Ensuite, pour les autres membres, puisque l'intrus n'a pas touché à l'exécution normale par rapport à ces membres, ils partagent la même clef $exp(exp(X_1, r_n), r_1) = exp(exp(X_i, r_n), r_i) = exp(exp(X_{n-1}, r_n), r_{n-1})$. L'intrus possède donc les vues des deux parties de groupe.

4.8 Conclusion

Nous avons présenté dans ce chapitre une étude qui s'applique aux protocoles de groupe, et plus généralement aux protocoles contribuants, i.e. les protocoles où les participants contribuent ensemble afin d'atteindre à un but (par exemple, générer une clef de session). Elle permet de

modéliser les protocoles eux-mêmes, mais surtout de décrire leurs caractéristiques ainsi que les propriétés de sécurité qu'ils doivent satisfaire. Notre objectif était de considérer une classe plus générale que celle des protocoles GDH. La formalisation présentée permet d'éviter toute ambiguïté sur la définition des problèmes de sécurité posés, et donc de garantir que les recherches de failles s'effectuent sur de bonnes bases. Nous avons illustré ces travaux sur quatre protocoles connus défectueux en se basant sur des scénarii d'attaques. Les protocoles étudiés sont les protocoles A-GDH.2 et SA-GDH.2, Asokan-Ginzboorg et Bresson-Chevassut-Essiari-Pointcheval.

Nous avons également présenté une stratégie de recherche d'attaques qui s'applique aux protocoles de groupe. Cette stratégie combine le modèle de [101] avec le modèle de services. En effet, elle utilise le modèle de services pour déduire les contraintes relatives à la violation de la ou des propriétés de sécurité à vérifier. Ces contraintes sont utilisées avec les contraintes du protocole pour obtenir une trace d'exécution d'une éventuelle attaque.

Nous avons appliqué notre stratégie sur trois protocoles : le protocole Asokan-Ginzboorg et les protocoles GDH.2 et A-GDH.2. Nous avons trouvé une attaque d'accord de clefs pour le protocole de Asokan-Ginzboorg. Cette attaque considère une exécution de deux sessions parallèles à deux participants chacune. Pour le protocole GDH.2, pour une exécution à quatre participants, nous avons trouvé deux attaques d'authentification. La première est par rapport au troisième participant et la deuxième attaque est par rapport au quatrième participant. Ces deux attaques ont été généralisées pour construire des attaques à n participants. Concernant le protocole A-GDH.2, pour une exécution à quatre participants, l'intrus arrive à diviser le groupe en deux sous-groupes ayant chacun une vue de la clef différente de l'autre. En plus, l'intrus possède ces deux vues de clefs. Cette attaque est aussi généralisée pour une attaque à n participants.

La méthode de falsification de protocoles proposée peut donc chercher d'éventuelles attaques concernant les différentes propriétés de sécurité des protocoles de groupe définies par le modèle de services. Néanmoins, bien que la plupart des attaques trouvées pour les protocoles étudiés ont pu être généralisées à n participants, l'entrée de notre méthode, comme la plupart des techniques d'autres outils de recherche d'attaques, reste toujours des instances du protocole étudié avec un nombre fini de participants. Cependant, la fixation du nombre de participants à l'avance fait perdre d'éventuelles attaques. En plus, comme toute méthode de falsification de protocoles en général, l'absence d'attaques pour un protocole étudié ne signifie pas que le protocole est correct. Le but du chapitre suivant est de proposer une méthode pour vérifier les protocoles de groupe pour un nombre quelconque de participants.

5

Un modèle synchrone pour la vérification de protocoles paramétrés

Sommaire

5.1 Introduction

Nous nous sommes intéressés dans le Chapitre 4, à la falsification de protocoles de groupe vis-à-vis d'une variété de propriétés de sécurité. Nous avons pu trouver des failles pour différents

protocoles de groupe par application de notre méthode de recherche d'attaques. Cependant, notre procédure avait comme entrée des instances bien particulières de protocoles où le nombre de participants est fixé d'avance. Notre méthode peut donc manquer des attaques nécessitant plus de participants. Par conséquent, ne pas trouver une attaque par notre méthode n'entraîne pas la correction du protocole.

Or, nous avons présenté également en chapitre 3 les problèmes que peut poser la vérification des protocoles de groupe. En particulier, mis à part leurs propriétés de sécurité sophistiquées, et plus spécifiquement pour les protocoles d'accord de clefs, la clef de groupe se base essentiellement sur les contributions de tous les membres du groupe. La modélisation de ces protocoles nécessite donc le traitement de listes non bornées. Leur vérification se confronte alors au fait que ces protocoles peuvent avoir un nombre arbitrairement important d'étapes vu que le nombre de participants est à priori non borné. Ensuite, certains participants tels que les serveurs ou les leaders manipulent des listes d'informations provenant des autres participants. Ils se trouvent donc contraints à traiter ces listes de manière récursive ou itérative afin de récupérer les contributions des autres participants ou de construire des messages destinés à chacun d'eux.

Toutes ces caractéristiques des protocoles de groupe permettent de coder facilement des problèmes indécidables. Cependant, comme détaillé en chapitre 3, peu de travaux ont traité ces problématiques, à savoir des protocoles avec nombre non borné de participants et avec des calculs récursifs. L'analyse formelle d'un tel protocole remonte à Paulson [77] avec une étude du protocole RA (*Recursive Authentication*) [23]. Cependant, si le protocole est défectueux, il n'y a pas de mécanisme automatique permettant de retrouver l'attaque correspondante. D'autres travaux ont vu le jour récemment. Cependant, certains travaux [59, 60, 99] ont utilisé uniquement des clefs atomiques. D'autres n'ont considéré que le cas d'un intrus passif [57].

Le but de ce chapitre est de proposer un modèle synchrone (voir [27] et [28]) pour les protocoles paramétrés avec un nombre fini de sessions. Ce modèle est une extension de modèles classiques de vérification de protocoles tels que [89] afin de pouvoir manipuler des listes de messages dont la longueur est donnée comme paramètre. Mis à part les protocoles de groupe, la manipulation de listes existe aussi dans différents domaines. Nous pouvons citer par exemple les protocoles de web services où des listes de messages sont souvent utilisées (voir exemple 5.1.0.1).

Exemple 5.1.0.1 *Le besoin de manipuler des listes se ressent aussi pour ce cas très basique d'opération effectuée par un participant honnête. Ce participant reçoit une liste $m_1, .., m_n$ de messages dont la longueur n n'est pas connue à l'avance et répond par l'envoi d'un message construit à base des informations qu'il vient de recevoir :*

$$\langle \{m_1\}_k, .., \{m_n\}_k \rangle \longrightarrow f(\langle m_1, .., m_n \rangle)$$

Dans cette étape, le participant décrypte les différents messages de la liste qu'il vient de recevoir en utilisant la clef publique de l'émetteur. Il construit après le message à envoyer en utilisant la liste $m_1, .., m_n$ de messages récupérés et la fonction f.

Notons que l'action basique présentée dans l'Exemple 5.1.0.1 est utilisée notamment dans les services web où les messages peuvent contenir des listes de nœuds XML encryptés. En outre, notre modèle permet aussi de gérer les protocoles de groupe qui impliquent un nombre non borné de participants, et pouvant par la suite être vus comme des protocoles paramétrés par leur nombre de participants. Notons aussi que notre étude de cas de protocoles de groupe (le protocole Asokan-Ginzboorg) suit l'action de base de l'Exemple 5.1.0.1.

Ainsi, nous proposons un ensemble de règles d'inférence qui permet de vérifier la sécurité pour la classe de protocoles bien tagués avec clefs autonomes, et ceci en présence d'un intrus **actif**. Nous montrerons dans le Chapitre 6 que cet ensemble de règles est correct et complet. Nous montrerons dans le même chapitre que ces règles d'inférence terminent pour la classe de protocoles considérée, prouvant ainsi que notre système d'inférence fournit une procédure de décision pour cette classe. Cela signifie que, en considérant des protocoles paramétrés dont le nombre de participants n n'est pas fixé, si notre procédure finit sans trouver d'attaques, alors le protocole considéré n'admet aucune attaque pour toute valeur du paramètre n. Notre procédure permet aussi de traiter les protocoles avec **clefs composées**.

Nous présentons donc dans ce chapitre les ingrédients nécessaires pour la procédure de vérification de protocoles de groupes. Nous commençons tout d'abord par présenter notre étude de cas : le protocole de Asokan-Ginzboorg [6], qui illustrera les différentes notions introduites dans ce chapitre. Nous détaillons après dans la Section 5.3 l'idée derrière la considération de notre modèle de protocole qui sera présenté après dans la Section 5.4. Ce modèle se base essentiellement sur l'ajout d'un opérateur spécial nommé *mpair* pour pouvoir gérer les listes. La Section 5.3 présente donc le modèle synchrone qui correspond à la classe de protocoles de groupes considérée au départ. Cette section définit aussi la transformation effectuée pour un protocole d'origine afin de pouvoir le traiter dans notre modèle. Nous montrons dans la Section 5.6 que l'ajout naïf de l'opérateur *mpair* mène à l'indécidabilité du problème d'insécurité. D'où l'introduction de la classe de protocoles bien-tagués avec clefs autonomes. Ensuite, nous introduisons dans la Section 5.7 les prédicats auxiliaires et leur sémantique. Ces prédicats expriment la construction de messages à partir des connaissances de l'intrus. Ils sont utilisés pour former le système de contraintes dont la satisfaisabilité est équivalente à l'existence d'attaques pour le protocole considéré. Enfin, nous présentons dans la Section 5.8 l'ensemble de règles d'inférence applicable au système de contraintes défini dans la section précédente.

5.2 Étude de cas : Asokan-Ginzboorg

Nous considérons la même étude de cas que celle de la Section 4.6, à savoir le protocole de Asokan Ginzboorg qui sera illustré par l'Exemple 5.2.0.2. Nous avons choisi cet exemple vu qu'il présente les caractéristiques présentées en introduction du chapitre courant. En effet, ce protocole nécessite la présence parmi les participants d'un participant spécial qui est le leader. Ce participant récolte la liste des messages des autres participants et fabrique différents messages qui leur sont destinés en se basant sur les informations présentes dans la liste de messages qu'il vient de recevoir. Cette action suit l'action de base présentée par l'Exemple 5.1.0.1.

Exemple 5.2.0.2 *Le protocole de Asokan-Ginzboorg à $n+1$ participants.*
Ce protocole est 3-1-Running-Exampledonné avec plus de détails dans la Section 4.6. La séquence de messages échangés entre un leader P_{n+1} et un ensemble de participants $P_1, .., P_n$ est comme suit :

$$\begin{array}{lll} 1.P_{n+1} \longrightarrow \mathit{Tous} & : & a_{n+1}, \{e\}_p \\ 2.P_i \longrightarrow P_{n+1} & : & a_i, \{r_i, s_i\}_e,\ i{=}1,..,n \\ 3.P_{n+1} \longrightarrow P_i & : & \{\{s_j, j{=}1,..,n{+}1\}\}_{r_i},\ i{=}1,..,n \\ 4.P_i \longrightarrow P_{n+1} & : & a_i, \{s_i, h(s_1, .., s_{n+1})\}_k\ i = 1, .., n \text{ et } k = f(s_1, .., s_{n+1}). \end{array}$$

Dans cet échange de messages, e désigne une clef publique générée par le leader. Les s_i pour $i = 1..(n+1)$ sont des nonces représentant les contributions pour la clef de groupe des différents

participants y compris le leader. Quant aux a_i pour $i = 1..(n+1)$, ils désignent les identités des participants. Enfin, les r_i pour $i = 1..n$ représentent les clefs symétriques des participants (sauf le leader).

5.3 Du modèle asynchrone vers le modèle synchrone

Nous présentons dans cette section le modèle synchrone que nous considérons comme classe de départ pour notre procédure de décision adaptée aux protocoles de groupe.

5.3.1 Contexte

Nous examinons dans cette section la classe des protocoles d'établissement de clefs avec les deux grandes catégories suivantes. La première catégorie est celle d'accord de clefs où un nombre non borné de participants se met d'accord sur une clef de groupe pour sécuriser leurs communications. Cette clef se base sur les contributions de tous les membres du groupe. La deuxième catégorie est celle de la distribution de clefs où une entité centrale, qui est généralement le serveur, contrôle la génération et la mise à jour de la clef de groupe et la distribue à tous les membres du groupe.

Pour ces deux catégories et dans plusieurs cas de ce genre de protocoles, il existe deux types de participants. D'une part, il y a des participants *ordinaires* qui ont pratiquement les mêmes actions, i.e. ont le même comportement vis à vis des messages reçus et des messages envoyés en réponse. D'autre part, il y a des participants *spéciaux* qui ont des comportements différents des autres participants ordinaires. Nous pouvons citer l'exemple du leader ou encore du serveur. Un tel participant doit être capable de recevoir et traiter plusieurs messages provenant de différents participants. Il doit aussi pouvoir composer des messages basés sur les informations existant dans les messages qu'il vient de recevoir pour pouvoir envoyer les messages composés aux différents participants.

Nous nous intéressons à la communication entre l'ensemble $P_1, .., P_{n-1}$ des participants ordinaires et le leader ou le serveur P_n. Il existe deux situations dans ce contexte.

Première situation

Dans la première situation illustrée par la Figure 5.1, le leader communique avec tous les participants ordinaires. Il attend les contributions de tous ces participants. Il calcule après la clef de groupe ou une information pouvant générer cette clef pour enfin envoyer cette information (ou clef) à tous les participants. Ainsi, les communications sont restreintes, dans ce cas, aux communications entre le leader P_n et chacun des participants ordinaires P_i. Comme exemple de protocoles présentant ce genre de situations, nous pouvons citer le protocole de Asokan-Ginzboorg [6].

Deuxième situation

Dans la seconde situation illustrée par la Figure 5.2, le groupe est modélisé sous forme de chaîne où chaque participant ordinaire communique sa contribution à son voisin. Quant au dernier participant des participants ordinaires P_{n-1}, il communique le message composé par toutes les contributions de tous les participants précédents au leader. Ensuite, le leader envoie la clef du groupe ou une information qui mène à cette clef à tous les participants ou bien au dernier des participants ordinaires qui, lui, transfert cette information au reste du groupe. Comme exemples de tels protocoles, nous pouvons citer les protocoles GDH.2 [8] et RA [23].

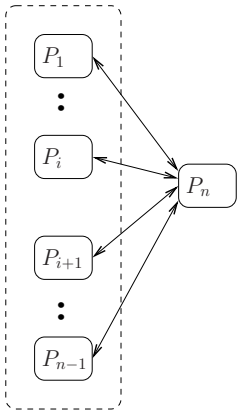


FIG. 5.1 – Protocoles de groupe : première situation

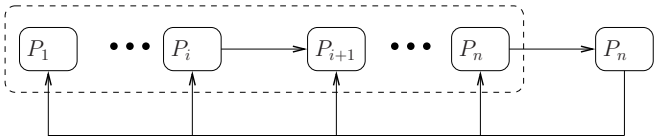


FIG. 5.2 – Protocoles de groupe : deuxième situation

5.3.2 Transformation du modèle asynchrone vers le modèle synchrone

L'idée de base de notre approche est de compresser les messages qui sont reçus par (ou envoyés à) plusieurs participants ordinaires, i.e. ayant des comportements similaires vis à vis d'un même message reçu. Ceci permet à un participant de recevoir tous les messages qu'il doit recevoir en une seule fois avant de les traiter pour composer le message à envoyer. Cette approche doit tenir compte du fait que, dans certains cas, le message envoyé est destiné à plusieurs participants au départ. Afin de pouvoir traiter cette problématique, nous proposons de réorganiser le protocole et plus précisément l'ensemble de participants. Nous proposons de représenter tous les participants ordinaires par un seul participant assez *spécial* que nous nommons le *simulateur*.

Nous nous intéressons à la première situation décrite dans le premier paragraphe de la Section 5.3.1. Nous nous focalisons sur la communication entre l'ensemble des participants ordinaires et le leader afin de déduire la transformation qui mène à une communication entre le leader et un *simulateur* qui simulerait l'ensemble de ces participants ordinaires.

Commençons par étudier le point de vue d'un participant P_i. Ce participant attend un message R_i et répond par le message S_i. Le message R_i provient nécessairement du leader. Nous distinguons deux cas. Dans le premier cas, le leader envoie la même information à tous les participants. Comme exemple de ce cas, nous pouvons citer l'envoi d'une clef à utiliser pour le chiffrement des contributions, comme c'est le cas du protocole Asokan-Ginzboorg lors de l'envoi du message $\{e\}_p$. Dans ce cas, la réception de chacun de ces messages peut être aussi vue comme la réception par le simulateur d'un seul message sous forme de liste composée à partir du même message reçu par tous les participants. Dans le second cas, le leader envoie une information différente à chacun des participants ordinaires. Le message qui serait reçu par le simulateur est alors une liste des messages $(R_1, .., R_{n-1})$ reçus par les différents participants qu'il simule. Le

message S_i est destiné au leader et contient généralement une contribution (une information privée) à la clef de groupe ou une confirmation d'une information déjà reçue. Dans ces deux cas, il existe une information privée pour chaque participant. Cependant, mis à part la ou les information(s) privée(s) du message à envoyer par un participant honnête, tous les messages ont la même forme, ou structure. Ainsi, si nous *simulons* cette phase d'envoi des différents messages S_i, $i = 1, .., (n-1)$ par les participants ordinaires, nous obtenons l'envoi, par le simulateur, d'un seul message $(S_1, .., S_{n-1})$. Nous illustrons dans la Figure 5.3 une étape du simulateur présentant la transformation de plusieurs étapes des participants ordinaires $P_1, .., P_{n-1}$, qui sont similaires modulo les informations privées de chacun des participants.

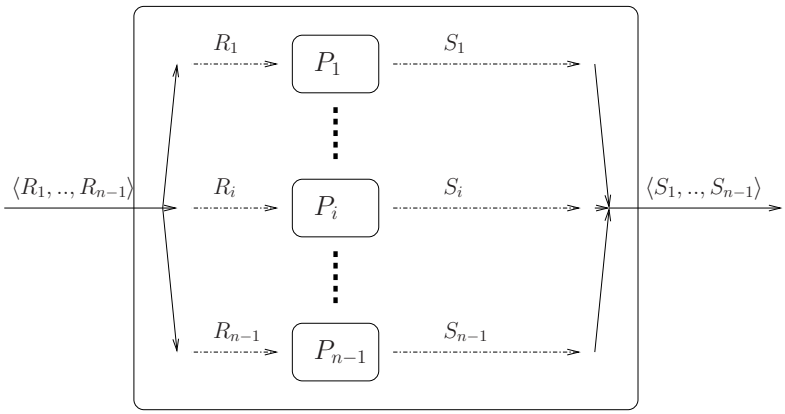


FIG. 5.3 – Une étape pour le simulateur dans le modèle synchrone

Nous nous focalisons maintenant sur le point de vue du leader P_n. Il reçoit différents messages R_{n_i} qui proviennent des participants ordinaires P_i. Cette action peut être vue comme étant la réception d'un seul message R_n sous forme d'une liste composée des messages reçus et donc de la forme $R_{n_1}, .., R_{n_{n-1}}$. Si le leader initialise la communication, alors le message qu'il envoie ne contient aucune information privée, ce qui correspond à un seul message envoyé à tout le monde. Cet acte peut être alors vu comme étant la réception, par le simulateur, d'une liste construite d'un même message (celui envoyé par le leader). Si, par contre, le leader compose un message destiné à chacun des participants et contenant des informations privées relatives à chacun d'eux, alors le message reçu par le simulateur serait une liste composée de tous ces messages. Nous illustrons dans la Figure 5.4 une étape pour le leader dans le nouveau modèle synchrone.

En résumé, nous avons le contexte suivant. D'une part, il y a un ensemble de participants $P_1, \ldots, P_{n-1}$ qui réagissent de manière similaire vis à vis d'un même message. Ils envoient leur contribution à la clef de groupe au leader (ou serveur). Ils reçoivent des informations du leader leur permettant de générer cette même clef. Une étape du simulateur simule donc l'ensemble des étapes décrivant la même action effectuée par les différents participants. D'autre part, le leader reçoit toutes les contributions en une seule fois du simulateur et envoie les messages destinés aux différents participants en une seule liste au simulateur. La transformation du modèle asynchrone en un modèle synchrone abstrait, donc de tous les participants ordinaires $P_1, .., P_n$ en un seul agent, est illustrée par la Figure 5.3.2.

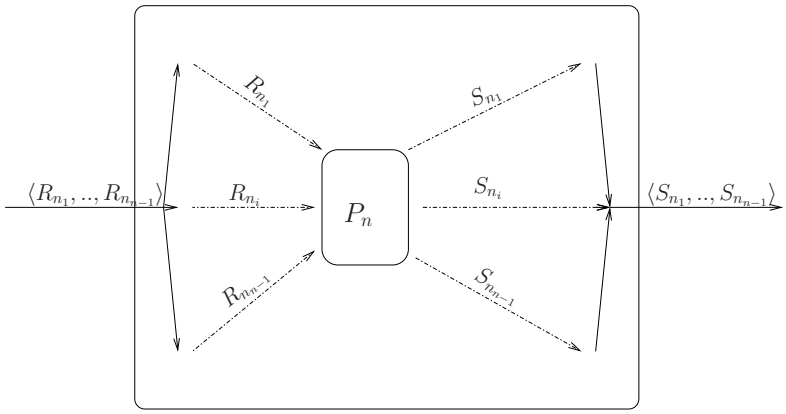


FIG. 5.4 – Une étape pour le leader dans le modèle synchrone

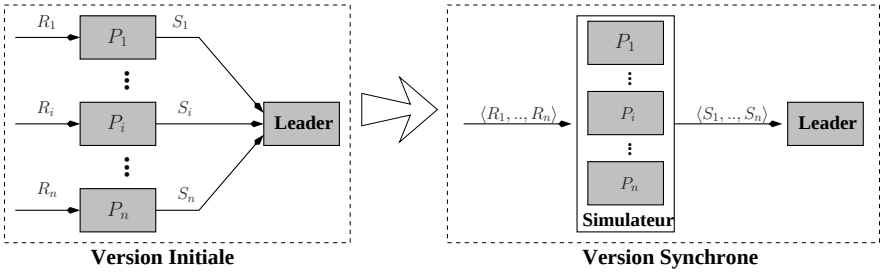


FIG. 5.5 – Transformation du modèle asynchrone en modèle synchrone

5.3.3 Application à notre étude de cas

Revenons à notre exemple : le protocole de Asokan-Ginzboorg. Nous présentons dans cette section la version synchrone de cet exemple. Le simulateur est noté par S et il simule les agents $P_1, \ldots, P_n$. L'agent L simule le leader P_{n+1}. La version synchrone est donnée par l'Exemple 5.3.3.1.

Exemple 5.3.3.1 *Dans cet exemple, nous présentons la version synchrone de notre étude de cas donnée par l'Exemple 5.2.0.2.*

$$
\begin{array}{lcll}
1.\ L & \longrightarrow & S & : \langle \langle l, \{e\}_p \rangle, .., \langle l, \{e\}_p \rangle \rangle \\
2.\ S & \longrightarrow & L & : \langle \langle a_1, \{\langle r_1, s_1 \rangle\}_e \rangle, .., \langle a_n, \{\langle r_n, s_n \rangle\}_e \rangle \rangle \\
3.\ L & \longrightarrow & S & : \langle \{\langle \langle s_1, .., s_n \rangle, s' \rangle\}_{r_1}, .., \{\langle \langle s_1, .., s_n \rangle, s' \rangle\}_{r_n} \rangle \\
4.\ S & \longrightarrow & L & : \langle \langle a_1, \{\langle s_1, h(\langle \langle s_1, .., s_n \rangle, s' \rangle) \rangle\}_{f(\langle \langle s_1, .., s_n \rangle, s' \rangle)} \rangle, .., \\
 & & & \quad \langle a_n, \{\langle s_n, h(\langle \langle s_1, .., s_n \rangle, s' \rangle) \rangle\}_{f(\langle \langle s_1, .., s_n \rangle, s' \rangle)} \rangle \rangle
\end{array}
$$

où l est l'identité du leader qui est dans la version initiale a_{n+1}, et s' sa contribution. Notons que chaque composante du message envoyé à l'étape 3 est formé de deux parties : la première partie correspond à l'ensemble des contributions des participants. Quant à la deuxième partie, elle correspond à la contribution du serveur.

5.4 Modèle de protocole

Nous étendons le modèle défini dans [89] afin de gérer les listes paramétrées par leur longueur. Ces listes sont construites par le biais d'un nouvel opérateur noté $mpair(\_,\_)$. L'intuition derrière cet opérateur est de générer une liste de messages à partir d'un pattern et d'une variable d'indice dont on fait varier la valeur.

5.4.1 Termes, taggage et indices

Soit $\mathcal{X}$ l'ensemble des variables représentées par des lettres majuscules. Soit $\mathcal{I}$ l'ensemble dénombrable de variables d'indices. Soit $\overrightarrow{\mathcal{X}}$ l'ensemble, disjoint de $\mathcal{X}$, de symboles représentés par des lettres majuscules sur-fléchées. Nous désignons par $\mathcal{X}_\mathcal{I} = \{Y_i \,|\, \overrightarrow{Y} \in \overrightarrow{\mathcal{X}} \text{ et } i \in \mathcal{I}\}$ l'ensemble dénombrable de variables indicées. Notons ici que si $Y_i \in \mathcal{X}_\mathcal{I}$, alors $Y \notin \mathcal{X}$.

D'une manière similaire, soient $\mathcal{C}$ et $\overrightarrow{\mathcal{C}}$ les deux ensembles de symboles représentés par des lettres minuscules sur-fléchées. Soit $\mathcal{C}_\mathcal{I} = \{c_i \,|\, \overrightarrow{c} \in \overrightarrow{\mathcal{C}} \text{ et } i \in \mathcal{I}\}$. Les éléments de $\mathcal{C}$ et $\mathcal{C}_\mathcal{I}$ sont appelés des constantes. Les termes peuvent être tagués (optionnellement) par un indice. Pour cela, nous réservons le symbole $\overrightarrow{e} \in \overrightarrow{\mathcal{C}}$ uniquement pour les opérations de taggage. Ainsi, un terme est un élément de $\mathcal{T}$ dans le langage suivant :

$$\begin{array}{rcl} \mathcal{T}_s & = & \{\mathcal{T}\}^p_\mathcal{T} \,|\, \{\mathcal{T}\}^s_\mathcal{T} \,|\, h(\mathcal{T}) \,|\, \langle \mathcal{T}, \mathcal{T} \rangle \,|\, mpair(\mathcal{I}, \mathcal{T}) \,|\, \mathcal{X} \,|\, \mathcal{X}_\mathcal{I} \,|\, \mathcal{C} \,|\, \mathcal{C}_\mathcal{I} \backslash \{e_i \,|\, i \in \mathcal{I}\} \\ \mathcal{T} & = & [e_i, \mathcal{T}_s] \,|\, \mathcal{T}_s \qquad \text{avec } i \in \mathcal{I} \end{array}$$

$\mathcal{T}_s$ est par définition l'ensemble des termes non tagués. Les opérateurs $\{\_\}^p_\_$ et $\{\_\}^s_\_$ représentent respectivement les chiffrements asymétriques et symétriques. L'opérateur $\langle \_, \_ \rangle$ représente la concaténation et h est une fonction de hachage. Notons H l'ensemble de fonctions de hachage. Quant à l'opérateur $mpair(i,t)$, il représente la liste (ou tuple) de termes construite à partir d'une forme commune t et ceci en itérant l'indice i parmi les entiers. La fonction de remplacement (voir Définition 5.4.1.3) définit la sémantique de cet opérateur, i.e. son interprétation dans le modèle standard de Dolev-Yao. Cette fonction remplace tout *mpair* par une séquence d'applications de *pair*. Le nombre de telles applications est donné comme paramètre (e) à la fonction. Cet entier représente la longueur commune des listes des termes modélisées par tout $mpair(,)$.

Avant de donner la fonction de remplacement dans Définition 5.4.1.3, nous définissons deux types d'opérations d'indices : les remplacements utilisés aussi dans les règles d'inférence sur les contraintes, et les substitutions utilisées pour la définition des solutions des contraintes (voir Section 5.7).

Définition 5.4.1.1 *Remplacement d'indices δ, substitution d'indices τ.*
Un remplacement d'indices δ (resp. une substitution d'indices τ) est une application de $\mathcal{I}$ à $\mathcal{I}$ (resp. aux entiers non négatives) qui est étendue aux variables et constantes indicées par : $\delta(X_i) = X_{\delta(i)}$ et $\delta(c_i) = c_{\delta(i)}$ (resp. $\tau(X_i) = X_{\tau(i)}$ et $\tau(c_i) = c_{\tau(i)}$). Le remplacement ainsi que la substitution d'indices sont aussi étendus par homomorphisme aux termes et aux ensembles de termes.

Nous utilisons aussi les notations $\delta_{i,j}$ (resp. $\tau_{i,j}$) pour désigner le remplacement (resp. la substitution) de $i \in \mathcal{I}$ par $j \in \mathcal{I}$ (resp. $j \in \mathbb{N}$). Nous utilisons aussi $\delta^k_{i,j}$ pour désigner le remplacement de $i \in \mathcal{I}$ par $j \in \mathcal{I}$ et les autres indices différents de i par $k \in \mathcal{I}$. Nous étendons cette notation aux ensembles d'indices avec la notion de $\delta^k_{Q,j}$ pour un ensemble $Q \subseteq \mathcal{I}$ pour désigner le remplacement de tout indice de Q par j et tout autre élément n'appartenant pas à Q par k.

Exemple 5.4.1.2 *Remplacement, substitution d'indices.*
Soient $i, j, k, l, m \in \mathcal{I}$, $c_i, c_j \in \mathcal{C}_\mathcal{I}$ et $X_i \in \mathcal{X}_\mathcal{I}$. Nous donnons ci-dessous quelques exemples d'application de remplacement et de substitution d'indices.

$$\begin{array}{l}\delta_{i,j}(c_i) = c_j \\ \tau_{i,2}(c_i) = c_2 \\ \tau^l_{i,k}(\{X_i\}_{c_j}) = \{X_k\}_{c_l} \\ \tau^l_{\{i,j\},k}(X_i) = X_k\end{array}$$

Nous définissons maintenant la fonction de remplacement comme suit :

Définition 5.4.1.3 *Remplacement de termes.*
Soit $\mathcal{T}_{DY}$ l'ensemble des termes sans aucun $mpair(\_,\_)$. Étant donné un entier e, la fonction de remplacement de termes $^{-e}$ de $\mathcal{T}$ à $\mathcal{T}_{DY}$ est définie ci-dessous :

$$\begin{array}{lcl}\overline{mpair(i,t)}^e & = & \langle\overline{\tau_{i,1}(t)}^e, .., \overline{\tau_{i,e}(t)}^e\rangle \\ \overline{f(s_1,..,s_k)}^e & = & f(\overline{s_1}^e, .., \overline{s_k}^e), \text{ pour tout } f \neq mpair\end{array}$$

Nous désignons par la signature $\mathcal{G}$ l'ensemble des opérateurs sur $\mathcal{T}_s$. Pour simplifier la syntaxe, dans tout ce qui suit, nous écrivons t^i au lieu de $[e_i, t]$, et nous l'appelons *terme tagué*. Nous omettons aussi le tag i pour le terme t^i quand le tag i n'est pas relevant à la discussion. Nous désignons par $\mathcal{T}_g$ l'ensemble de termes clos, i.e. tout terme $t \in \mathcal{T}$ sans variable dans $\mathcal{X}$ ou $\mathcal{X}_\mathcal{I}$ et sans symbole $mpair$. Les termes clos seront utilisés pour décrire les messages circulant dans une exécution du protocole. Nous les appelons aussi des messages. Étant donné un terme t, nous dénotons par $Var(t)$ (resp. $Cons(t)$) l'ensemble de variables (resp. constantes) existant dans t. Nous dénotons par $Atoms(t)$ l'ensemble $Var(t) \cup Cons(t)$.

Afin de représenter une liste de termes, nous itérons l'opérateur de concaténation $\langle\_,\_\rangle$. Par exemple, pour représenter a, b, c, d nous pouvons utiliser le terme $\langle a, \langle b, \langle c, d\rangle\rangle\rangle$. Dans ce cas, nous le notons $\langle a, b, c, d\rangle$. Cependant, nous ne supposons aucune propriété d'associativité de l'opérateur de concaténation.

Exemple 5.4.1.4 *Le protocole Asokan-Ginzboorg synchrone.*
Nous revenons à la version synchrone du protocole Asokan-Ginzboorg présenté par l'Exemple 5.3.3.1. Nous modélisons ce protocole dans notre modèle en remplaçant toute liste $\langle m_1, .., m_n\rangle$ de messages, ayant la même forme (structure), par le message $mpair(i, m_i)$.

$$\begin{array}{lcll}1.\ L & \longrightarrow & S & : mpair(i, \langle l, \{e\}_p\rangle) \\ 2.\ S & \longrightarrow & L & : mpair(i, \langle a_i, \{\langle r_i, s_i\rangle\}_e\rangle) \\ 3.\ L & \longrightarrow & S & : mpair(i, \{\langle mpair(j, s_j), s'\rangle\}_{r_i}) \\ 4.\ S & \longrightarrow & L & : mpair(i, \langle a_i, \{\langle s_i, h(\langle mpair(k, s_k), s'\rangle)\rangle\}_{f(\langle mpair(k,s_k),s'\rangle)}\rangle)\end{array}$$

Une substitution σ assigne des termes aux variables. Une substitution close assigne des termes clos aux variables. L'application de σ pour un terme t est notée $t\sigma$. Ces notations sont étendues aux ensembles de termes E d'une manière standard : $E\sigma = \{t\sigma | t \in E\}$. Nous désignons par $STermes(t)$ l'ensemble de sous-termes de t et par $dpth(t)$ la profondeur de t. Ces deux ensembles sont définis récursivement comme suit : Si t est une variable ou constante alors $STermes(t) = \{t\}$ et $dpth(t) = 1$. Si $t = f(t_1, .., t_n)$ ou $t = f(t_1, .., t_n)^i$ avec $t \in \mathcal{G}$, alors $STermes(t) = \{t\} \cup \cup_{i=1}^n STermes(t_i)$ et $dpth(t) = 1 + MAX_{i=1..n}\{dpth(t_i)\}$. Notons que u n'est pas considéré comme sous-terme de u^i. Nous désignons par $\leq$ la relation de sous-terme sur $\mathcal{T}$.

Nous définissons la relation $\leqslant_m$ pour $\mathcal{T} \times \mathcal{T}$ comme la plus petite relation réflexive et transitive telle que si $t = f(t_1, .., t_n)$ ou $t = f(t_1, .., t_n)^j$ avec $f \neq mpair$, alors pour tout $i = 1, \ldots, m$ nous avons $t_i \leqslant_m f(t_1, .., t_m)$. Notons que $t \leqslant_m u$ implique $t \leq u$.

Finalement, nous définissons l'ensemble d'indices se produisant dans un terme comme suit :

Définition 5.4.1.5 *Indices de termes.*
Soit un terme $t \in \mathcal{T}$. Nous désignons par $Var_{\mathcal{I}}(t)$ l'ensemble d'indices dans t défini récursivement comme suit :

$$\begin{array}{ll} Var_{\mathcal{I}}(mpair(i,t)) = Var_{\mathcal{I}}(X) = Var_{\mathcal{I}}(c) = \emptyset & \text{avec } X \in \mathcal{X} \text{ et } c \in \mathcal{C} \\ Var_{\mathcal{I}}(X_i) = Var_{\mathcal{I}}(c_i) = \{i\} & \text{avec } X_i \in \mathcal{X}_{\mathcal{I}} \text{ et } c_i \in \mathcal{C}_{\mathcal{I}} \\ Var_{\mathcal{I}}(f(t_1, .., t_n)) = Var_{\mathcal{I}}(t_1) \cup .. \cup Var_{\mathcal{I}}(t_n) & \text{avec} f \neq mpair \\ Var_{\mathcal{I}}(t^i) = Var_{\mathcal{I}}(t) \cup \{i\} & \end{array}$$

Nous définissons aussi l'ensemble d'indices de variables dans $u \in \mathcal{T}$ comme étant :

$$Var^{\mathcal{X}}_{\mathcal{I}}(u) = Var_{\mathcal{I}}(u) \cap Var_{\mathcal{I}}((STermes(u) \cap \mathcal{X}_{\mathcal{I}}))$$

Exemple 5.4.1.6 *Indices de termes, indices de variables.*

$$\begin{array}{l} Var_{\mathcal{I}}(c_i) = \{i\} \\ Var^{\mathcal{X}}_{\mathcal{I}}(c_i) = \emptyset \\ Var_{\mathcal{I}}(X_i) = Var^{\mathcal{X}}_{\mathcal{I}}(X_i) = \{i\} \\ Var_{\mathcal{I}}(\{\langle c_i, mpair(j, X_j)\rangle\}_{{Y_k}^k}) = Var_{\mathcal{I}}(c_i) \cup Var_{\mathcal{I}}(mpair(j, X_j)) \cup Var_{\mathcal{I}}({Y_k}^k) \\ \qquad\qquad = \{i, k\} \\ Var^{\mathcal{X}}_{\mathcal{I}}(\{\langle c_i, mpair(j, X_j)\rangle\}_{{Y_k}^k}) = \{i, k\} \cap \{j, k\} = \{k\} \end{array}$$

5.4.2 Spécification du protocole

Un protocole est modélisé par un ensemble de participants et un ensemble fini d'étapes pour chacun d'eux. Nous associons à chacun des participants A un ensemble partiellement ordonné d'étapes représenté par $(W_A, <_{W_A})$ où W_A sont les indices des étapes de A et $<_{W_A}$ est un ordre partiel sur W_A. Chaque étape est de la forme $R_\iota \Rightarrow S_\iota$ où R_ι est un message attendu par A et S_ι désigne la réponse correspondante. *Init* et *End* sont des messages fixés utilisés pour initier et clôturer une session du protocole. Un protocole P est donc défini comme étant égal à $\{R_\imath \Rightarrow S_\imath | \imath \in J\}$ où J est un ensemble d'indices d'étapes qui sont partiellement ordonnés. Notre notion d'exécution correcte de session de protocole (ou *exécution de protocole*) suit [89].

Exemple 5.4.2.1 *Revenons à notre étude de cas : le protocole Asokan-Ginzboorg. La spécification de la version synchrone de ce protocole décrite par l'Exemple 5.4.1.4 est donnée ci-dessous :*

$$\begin{array}{lll} (L,1) & Init & \Rightarrow \quad mpair(t, \langle l, \{e\}_p\rangle) \\ (S,1) & mpair(i, \langle L, \{E_i\}_p\rangle) & \Rightarrow \quad mpair(j, \langle a_j, \{\langle r_j, s_j\rangle\}_{E_j}\rangle) \\ (L,2) & mpair(k, \langle a_k, \{\langle R_k, S_k\rangle\}_e\rangle) & \Rightarrow \quad mpair(m, \{\langle mpair(o, S_o), s'\rangle\}_{R_m}) \\ (S,2) & \quad mpair(q, \{\langle mpair(u, s_u), S'\rangle\}_{r_q}) & \Rightarrow \\ & mpair(w, \langle a_w, \{\langle s_w, h(\langle mpair(y, s_y), S'\rangle)\rangle\}_{f(\langle mpair(y,s_y),S'\rangle)}\rangle) & \\ (L,3) & mpair(x, \langle a_x, \{\langle S_x, h(\langle mpair(z, S_z), s'\rangle)\rangle\}_{f(\langle mpair(z,S_z),s'\rangle)}\rangle) \Rightarrow End & \end{array}$$

L'ensemble ordonné des étapes de chacun des participants est : $W_L = 1, 2, 3$ et $W_S = 1, 2$ avec l'ordre $1 <_{W_L} 2 <_{W_L} 3$, et $1 <_{W_S} 2$.

5.4.3 Modèle de l'intrus

Nous suivons toujours le modèle de l'intrus de Dolev-Yao [45]. Les actions de l'intrus sont simulées par une séquence de règles de réécriture sur l'ensemble des termes. Ces règles regroupent des règles de décomposition et de composition. Elles sont définies ci-dessous.

Règles de décomposition	**Règles de composition**
$\langle a_1, \ldots, a_n \rangle \rightarrow a_1, \ldots, a_n, \langle a_1, \ldots, a_n \rangle$	$a_1, \ldots, a_n \rightarrow a_1, \ldots, a_n, \langle a_1, \ldots, a_n \rangle$
$\{a\}_K^p, K^{-1} \rightarrow \{a\}_K^p, K^{-1}, a$	$a, K \rightarrow a, K, \{a\}_K^p$
$\{a\}_b^s, b \rightarrow \{a\}_b^s, b, a$	$a, b \rightarrow a, b, \{a\}_b^s$
	$h, t \rightarrow h(t), h, t$ pour $h \in H$
$t^i \rightarrow t$	$t \rightarrow t^i$ pour $i \in \mathcal{I}$

Notons ici qu'il n'y a pas de règles de l'intrus pour l'opérateur *mpair* vu que la fonction de remplacement définie dans la Définition 5.4.1.3 donne l'interprétation de cet opérateur dans le modèle standard de Dolev-Yao : l'intrus ne traite que des paires de messages.

Nous notons chaque règle de composition d'un terme t par $L_c(t)$. De même, nous notons chaque règle de décomposition d'un terme t par $L_d(t)$.

Nous définissons la relation de réécriture, notée par $M \rightarrow M'$ s'il existe une règle $l \rightarrow r$ parmi les règles de composition et de décomposition de l'intrus telle que, l est un sous terme de M et M' est obtenu en remplaçant l par r dans M. Nous désignons par $E \rightarrow_R E'$ l'application d'une règle R à un ensemble de messages E ayant pour résultat E'. Nous notons la clôture réflexive et transitive de $\rightarrow$ par $\longrightarrow_{DY}^*$.

5.4.4 Dérivations, attaques

Nous suivons la même définition de dérivation de [89]. Nous appelons donc une dérivation D une séquence d'applications de règles $E_0 \rightarrow_{R_1} \ldots \rightarrow_{R_n} E_n$. Les règles R_i pour $i = 1, .., n$ sont appelées les règles de la dérivation D. Nous notons $R \in D$ pour dire que R est l'une des règles $R_i, i = 1, .., n$ utilisées dans D. Nous notons aussi $D_t(E)$ la dérivation qui a pour but de dériver t à partir de E.

Nous définissons le prédicat Dy. Ce prédicat vérifie si un message peut être construit par l'intrus à partir d'un certain ensemble de messages qu'il connaît.

Définition 5.4.4.1 *Dy, Dy_c and Dy_d.*
Soient E et $\mathcal{K}$ deux ensembles de termes clos et t un terme clos tels que il existe une dérivation ayant pour but t sans utiliser aucun terme de $\mathcal{K}$ comme clef de déchiffrement. Alors, nous disons que t est forgé à partir de E et nous le notons $t \in Dy(E, \mathcal{K})$. Ensuite, si $D = D'.L_c(t)$ alors $t \in Dy_c(E, \mathcal{K})$, sinon $t \in Dy_d(E, \mathcal{K})$.

Nous pouvons maintenant définir la notion d'attaque pour un protocole dans notre modèle. Cette définition se base sur le prédicat Dy.

Définition 5.4.4.2 (Attaque) *Étant donné un protocole $P = \{R'_\imath \Rightarrow S'_\imath | \imath \in J\}$, un secret Sec et en supposant que l'intrus a comme connaissances initiales S_0, une attaque est décrite par une substitution close σ, un entier e, et un ordre d'exécution correct $\pi : J \longrightarrow 1, \ldots, k$ t.q. $\forall i = 1, \ldots, k$, nous avons :*

$$\overline{R_i}^e \sigma \in Dy(\{\overline{S_0}^e, \overline{S_1}^e \sigma, \ldots, \overline{S_{i-1}}^e \sigma\}, \emptyset)$$
$$\overline{Sec}^e \in Dy(\{\overline{S_0}^e, \overline{S_1}^e \sigma, \ldots, \overline{S_k}^e \sigma\}, \emptyset)$$

avec $R_i = R'_{\pi^{-1}(i)}$ *et* $S_i = S'_{\pi^{-1}(i)}$.

D'après cette définition, une exécution correspond à une attaque si :
- cette exécution peut être exécutée. En effet, pour chaque étape de l'exécution, l'intrus doit être capable de forger le message attendu par le participant à partir des connaissances qu'il a acquises tout au long des étapes précédentes.
- à la fin de cette exécution, l'intrus doit être capable de forger le secret Sec à partir des connaissances acquises durant toute l'exécution.

Notons que toutes les listes ont toujours la même longueur e. Ainsi, pour les protocoles de groupe, nous supposons que le groupe a toujours le même nombre de membres.

5.5 Équivalence entre le modèle asynchrone et le modèle synchrone

Nous revenons à la transformation entre les modèles asynchrone et synchrone décrite dans la Section 5.3 pour nous intéresser à l'équivalence entre ces deux modèles. Par construction, la transformation du modèle asynchrone vers le modèle synchrone est correcte. En effet, si nous trouvons une attaque dans le modèle synchrone alors il s'agit bien d'une vraie attaque dans le modèle initial (asynchrone). Quant à la complétude de cette transformation, elle représente le fait de ne pas perdre d'éventuelles attaques : s'il existe une attaque dans le modèle initial, alors il existe une attaque correspondante dans le modèle synchrone. La complétude de cette transformation n'est pas prouvée d'une manière générique (pour tout protocole) vu qu'elle dépend des caractéristiques de chaque protocole.

Le but de cette section est de donner une idée de la preuve d'équivalence entre les modèles synchrone et asynchrone. Nous l'appliquons après sur notre étude de cas.

L'idée de base derrière la transformation est de représenter un certain nombre de participants $P_1, .., P_n$ par un seul participant qui est le simulateur et de simuler leur comportement. C'est ainsi que, dans le modèle synchrone une étape de la forme $R_i \Rightarrow S_i$ effectué par le simulateur remplace un ensemble d'étapes effectuées par chacun des participants : il remplace $R_{i_1} \Rightarrow S_{i_1}, .., R_{i_n} \Rightarrow S_{i_n}$ avec $R_i = \langle R_{i_1}, .., R_{i_n} \rangle$ et $S_i = \langle S_{i_1}, .., S_{i_n} \rangle$.

Si nous nous mettons maintenant à la place de l'intrus, son objectif est d'assurer l'exécutabilité du protocole et par la suite de construire à chaque étape le message attendu par le participant qui est R_i dans le modèle synchrone et R_{i_j} dans le modèle asynchrone pour l'étape décrite ci-dessus. La question que doit poser cette transformation est alors, est-il possible pour l'intrus d'utiliser une information envoyée par un participant P_j (S_{i_j}) pour composer un message destiné à un participant P_k (R_{i_j}) dans la même étape $R_i \Rightarrow S_i$ du simulateur ? Si c'est le cas, alors la transformation n'est pas complète, sinon, elle l'est.

Soit e la valeur de n. La condition de non complétude devient alors, suivant la définition d'exécutabilité de la Section 5.4.4 :

$$\exists j \in \{1, .., e\} \text{ t.q. } \overline{R_{i_j}}^e \sigma \in Dy(\{\overline{S_0}^e, \overline{S_1}^e \sigma, \ldots, \overline{S_{i-1}}^e \sigma, \overline{S_{i_1}}^e \sigma, \ldots, \overline{S_{i_{j-1}}}^e \sigma\}, \emptyset)$$

Exemple 5.5.0.3 *Équivalence entre les deux modèles pour Asokan-Ginzboorg.*
Nous considérons la spécification du protocole Asokan-Ginzboorg présentée par l'Exemple 5.4.2.1. Nous étudions l'équivalence entre les deux modèles (synchrone et asynchrone) pour ce protocole et plus spécifiquement la complétude de notre transformation.

Dans la première étape, il n'y a rien à prouver puisque Init est un message qui est déjà connu par l'intrus.

Nous allons vérifier dans la deuxième phase ($(S,1)$ de l'Exemple 5.4.2.1), si le message $\langle a_j, \{\langle r_j, s_j\rangle\}_{E_j}\rangle$ récupéré d'un participant P_j peut servir à composer le message $\langle L, \{E_i\}_p\rangle$ attendu par un autre participant P_i avec $i, j = 1..n$. Ceci implique que le message $\{E_i\}_p$ a été construit par unification avec le message $\{\langle r_j, s_j\rangle\}_{E_j}$ ce qui donne p comme valeur à E_j. Toutefois, pour le participant P_j, la valeur de E_j provient d'un message $\{E_j\}_p$ qu'il a reçu. Or, ceci est impossible puisque la clef p ne peut pas être déduite par l'intrus.

Dans la troisième étape ($(L,2)$), nous supposons que l'intrus a comme connaissances (modulo une substitution close σ) $\{l, \{e\}_p, mpair(i, \langle a_j, \{r_j, s_j\}_{E_j}\rangle)\}$. Nous vérifions si le message $\{s_1, .., s_n, s'\}_{R_l}$, récupéré d'un participant P_l, peut servir à composer le message $\langle a_k, \{R_k, S_k\}_e\rangle$, attendu par le participant P_k. Ceci est possible si nous unifions ces deux messages. Ceci implique que la variable R_l a pour valeur e. Or, la valeur de la variable R_l provient du message reçu $\{R_l, S_l\}_e$. Ainsi, soit ce message provient d'un message provenant d'un participant P_k ($\{s_1, .., s_n, s'\}_{R_l}$) et dans ce cas nous obtenons soit $e = s_1$ soit $e = s_1, .., s_i$ ce qui est impossible. Le deuxième cas est de l'unifier avec une des connaissances de l'intrus ($\{r_j, s_j\}_{E_j}$), et dans ce cas, nous obtenons $e = r_j$ ce qui est impossible.

Pour la quatrième étape ($(S,2)$), vu que, premièrement le message envoyé par chacun des participants est un hachage d'un message, et deuxièmement, le message attendu par un de ces participants P_q est chiffré par une constante r_q, il n'est possible donc ni d'unifier ces messages, ni d'utiliser un sous-terme du message envoyé parce que l'intrus n'a pas cette fonction de hachage.

Dans la cinquième étape $(L,3)$, il n'y a pas de messages envoyés par les participants.

Nous notons ici que la vérification de complétude de la transformation entre les deux modèles synchrones et asynchrones et par la suite l'équivalence entre ces modèles nécessite implicitement une analyse du protocole.

5.6 Résultat d'indécidabilité et restrictions

L'ajout naïf de l'opérateur *mpair* mène à un problème d'insécurité (existence d'une attaque) indécidable. Nous montrons ce résultat dans la Section 5.6.1 et nous déduisons les restrictions que nous adoptons en définissant ainsi une classe de protocoles paramétrés pour laquelle nous montrons la décidabilité dans le chapitre 6.

5.6.1 Indécidabilité du problème d'insécurité pour les protocoles paramétrés

Nous montrons l'indécidabilité du problème d'insécurité des protocoles paramétrés en codant le problème PCP (*Post Correspondance Problem*) à deux lettres. Nous notons que ce résultat utilise uniquement des clefs atomiques. Ce résultat exploite essentiellement l'autorisation des entrelacements (correspondance) entre les éléments d'une même liste. Nous commençons par définir le protocole PCP.

Définition 5.6.1.1 *Protocole PCP.*
Soit $J = \{(\alpha_1, \beta_1), .., (\alpha_p, \beta_p)\}$ une instance de PCP en considérant l'alphabet $\{a, b\}$.

Nous définissons une spécification du protocole $P(J)$ qui code J et qui utilise comme ensemble de constantes $\mathcal{C} = \{a, b, 0, t, u\}$, comme ensemble de variables $\mathcal{X} = \{Z\}$ et comme ensemble de vecteurs de variables $\overrightarrow{\mathcal{X}} = \left\{\overrightarrow{A}, \overrightarrow{B}, \overrightarrow{X}, \overrightarrow{Y}\right\}$. Dans cette spécification, nous utilisons uniquement

un seul participant honnête. Cette spécification est décrite comme suit :

$$
\begin{array}{lrcl}
1. & Init & \Rightarrow & a,\, b,\, 0,\, \{\langle 0,0\rangle\}_t \\
2. & mpair(i, \langle A_i, B_i\rangle) & \Rightarrow & mpair(i, \{\langle A_i, B_i\rangle\}_t) \\
3. & mpair(i, \{\langle X_i, Y_i\rangle\}_t) & \Rightarrow & mpair(i, \{\langle \alpha_1\,(X_i)\,, \beta_1\,(Y_i)\rangle\}_u), \ldots, \\
 & & & mpair(i, \{\langle \alpha_p\,(X_i)\,, \beta_p\,(Y_i)\rangle\}_u) \\
4. & mpair(i, \{\langle A_i, B_i\rangle\}_u), \{\langle Z, Z\rangle\}_u & \Rightarrow & Sec
\end{array}
$$

Le but de la première étape est de fournir à l'intrus l'alphabet avec un symbole de terminaison 0. Dans la deuxième étape, nous demandons à l'intrus de construire une liste de paires de mots en utilisant l'alphabet a, b qu'il a déjà reçu. Le reste de la spécification a pour but de tester si chacune de ces paires, choisies par l'intrus, peut être obtenue à partir d'une autre paire par extension de celle-ci par un des mots PCP (α_j, β_j).

Puisque la paire initiale de mots vides, représentée par $\{\langle 0,0\rangle\}_t$, ne peut pas être obtenue par extension d'une autre paire, elle est fournie à l'intrus à l'étape 1.

Dans la deuxième étape, nous fixons le choix de l'intrus (la liste des paires de mots) en encryptant chaque paire par la clef u. Une fois que ce choix des paires de mots est fait, l'intrus sélectionne quelques paires de mots (encryptées par t) obtenues à l'étape 2 et 1 ($\{\langle 0,0\rangle\}_t$). Il les reçoit encore une fois une à une, mais étendues par un des mots de l'instance de PCP, et encryptées par la clef u. Nous prévoyons que l'intrus sélectionne toutes les paires des mots qu'il a choisi à la deuxième étape sauf la plus longue qui sera remplacée par la paire des mots vides.

Finalement, à la quatrième étape, nous procédons à deux vérifications. La première consiste à vérifier que pour toute paire de mots choisie à l'étape 2, il existe une paire étendue de mots reçue à l'étape 3. Ainsi, nous vérifions par récurrence que chaque paire de termes choisie à l'étape 2 est une concaténation de mots de l'instance de PCP J, à partir de laquelle nous avons construit ce protocole. La deuxième vérification consiste à tester que l'une parmi les paires de mots attendues est une solution du problème PCP, i.e. une paire de mots identiques.

Théorème 5.6.1.2 *Une instance J de PCP admet une solution ssi il existe une attaque de secret Sec pour $P(J)$.*

La preuve de ce théorème découle des lemmes 5.6.1.3 (le sens indirect) et 5.6.1.4 (le sens direct).

Lemme 5.6.1.3 *Si pour un entier n il existe une exécution de $P(J)$ alors J admet une solution.*

Preuve. Nous supposons qu'il existe une exécution de $P(j)$ pour un certain entier n. Par définition du protocole, en suivant récursivement la création des paires étendues des étapes 2 et 3, pour chaque terme $\{\langle a, b\rangle\}_u$ connu par l'intrus, y compris $\{\langle A_j, B_j\rangle\}_u$ pour tout $j = 1, .., n$ ou $\{\langle Z, Z\rangle\}_u$, il existe $i \in 1..n$ et $k \in 1..p$ tels que $a = \alpha_k(X_i)$ et $b = \beta_k(Y_i)$. En effet, si nous regardons l'origine de $\{\langle a, b\rangle\}_t$, il a été créé à l'étape 2, et donc il existe un entier $j \in 1..n$ tel que $a = A_j$ et $b = B_j$. Or, l'étape de vérification (étape 4 et la création des termes $\{\langle A_j, B_j\rangle\}_u$) nous assure que pour chaque (A_j, B_j) il existe $i \in 1..n$ et $k \in 1..p$ tels que $A_j = \alpha_k(X_i)$ et $B_j = \alpha_k(Y_i)$. Ainsi, soit $X_i = Y_i = 0$ ($\{\langle a, b\rangle\}_t = \{\langle 0,0\rangle\}_t$), soit il existe $i' \in 1..n$ tel que $a = \alpha_k(A_{i'})$ *et* $b = \beta_k(B_{i'})$, et l'intrus connaît $\{\langle A_{i'}, B_{i'}\rangle\}_t$.

Par itération sur $A_{i'}$ et $B_{i'}$, en commençant par $\{\langle Z, Z\rangle\}_u$ et puisque $|\langle a, b\rangle| > |\langle A_{i'}, B_{i'}\rangle|$, il apparaît qu'il existe une liste $[j_1, .., j_r]$ d'indices de $1..p$ telle que $\alpha_{j_1}(\alpha_{j_2}(..\alpha_{j_r}(0)..)) = \beta_{j_1}(\beta_{j_2}(..\beta_{j_r}(0)..))$, et par la suite, $\alpha_{j_1}..\alpha_{j_r} = \beta_{j_1}..\beta_{j_r}$. Ainsi, J admet une solution. □

Lemme 5.6.1.4 *S'il existe une solution pour J alors il existe un entier n tel que $P(J)$ admet une exécution.*

PREUVE. Nous supposons qu'il existe une solution pour J. Soit donc $[j_1, .., j_r]$ une liste d'indices de $1..p$ telles que $\alpha_{j_1}..\alpha_{j_r} = \beta_{j_1}..\beta_{j_r}$. Nous choisissons r comme valeur de n ($n = r$).

Nous allons construire une exécution du protocole $P(J)$ de la manière suivante. Le résultat obtenu par l'étape 2 est une liste des $\{\langle A_i, B_i\rangle\}_t$ de la forme :

$$\langle\{\langle\alpha_{j_1}..\alpha_{j_n}(0), \beta_{j_1}..\beta_{j_n}(0)\rangle\}_t, .., \{\langle\alpha_{j_i}..\alpha_{j_n}(0), \beta_{j_i}..\beta_{j_n}(0)\rangle\}_t, .., \{\langle\alpha_n(0), \beta_n(0)\rangle\}_t\rangle$$

Notons que dans cette liste, nous pouvons déjà tester que la condition de l'étape de vérification ($\{\langle Z, Z\rangle\}_u$) est vérifiée puisque nous avons deux mots identiques ($\{\langle\alpha_{j_1}..\alpha_{j_n}(0), \beta_{j_1}..\beta_{j_n}(0)\rangle\}_t$) par hypothèse que $\alpha_{j_1}..\alpha_{j_r} = \beta_{j_1}..\beta_{j_r}$.

Le but après est d'arriver à construire les éléments de cette liste mais encryptés par la clef u. Pour ce faire, à l'étape 3, l'intrus choisit comme liste des $\{X_i, Y_i\}_t$ la liste suivante :

$$\langle\{\langle 0, 0\rangle\}_t, .., \{\langle\alpha_{j_i}..\alpha_{j_n}(0), \beta_{j_i}..\beta_{j_n}(0)\rangle\}_t, .., \{\langle\alpha_{j_n}(0), \beta_{j_n}(0)\rangle\}_t\rangle$$

Il obtient donc la liste suivante comme résultat de cette étape :

$$<\langle\{\langle\alpha_{j_1}(0), \beta_{j_1}(0)\rangle\}_u, \underline{\{\langle\alpha_{j_1}\alpha_{j_2}..\alpha_{j_n}(0), \beta_{j_1}\beta_{j_2}..\beta_{j_n}(0)\rangle\}_u}, .., \{\langle\alpha_{j_1}\alpha_{j_n}(0), \beta_{j_1}\beta_{j_n}(0)\rangle\}_u\rangle$$

...

$$\langle\{\langle\alpha_{j_i}(0), \beta_{j_i}(0)\rangle\}_u, .., \underline{\{\langle\alpha_{j_i}\alpha_{j_{i+1}}..\alpha_{j_n}(0), \beta_{j_i}\beta_{j_{i+1}}..\beta_{j_n}(0)\rangle\}_u}, .., \{\langle\alpha_{j_i}\alpha_{j_n}(0), \beta_{j_i}\beta_{j_n}(0)\rangle\}_u\rangle$$

...

$$\langle\underline{\{\langle\alpha_{j_n}(0), \beta_{j_n}(0)\rangle\}_u}, .., \{\langle\alpha_{j_n}\alpha_{j_i}..\alpha_{j_n}(0), \beta_{j_n}\beta_{j_i}..\beta_{j_n}(0)\rangle\}_u, .., \{\langle\alpha_{j_n}\alpha_{j_n}(0), \beta_{j_n}\beta_{j_n}(0)\rangle\}_u\rangle>$$

À la dernière étape, l'intrus construit la liste des $\{\langle A_i, B_i\rangle\}_u$ à partir des éléments qui sont soulignés ci-dessus. Nous retrouvons alors les mêmes composants de la liste du départ choisie par l'intrus à l'étape 2. En outre, par construction de cette liste, nous avons un élément qui y appartient et qui contient deux mots identiques ce qui vérifie le deuxième test de la dernière étape.

Nous allons maintenant construire cette exécution de manière formelle. Nous choisissons ces valeurs de σ pour les variables A_i, B_i :

$$\forall i \in 1..n-1,\ \sigma(A_i) = \alpha_{j_i}(\sigma(A_{i+1})) \text{ et } \sigma(B_i) = \beta_{j_i}(\sigma(B_{i+1}))$$
$$\sigma(A_n) = \alpha_{j_n}(0) \text{ et } \sigma(B_n) = \beta_{j_n}(0)$$

Cet ensemble de paires de mots est la liste choisie par l'intrus à l'étape 2. Ensuite, à l'étape 3, nous choisissons pour les variables X_i, Y_i ces valeurs :

$$\forall i \in 2..n,\ \sigma(X_i) = \sigma(A_i) \text{ et } \sigma(Y_i) = \sigma(B_i)$$
$$\sigma(X_1) = \sigma(Y_1) = 0$$

Pour ce choix, nous laissons les paires de mots choisies à l'étape 2 sauf pour la plus longue (la première) que l'on remplace par une paire de mots vides. Nous pouvons maintenant vérifier les deux tests de la dernière étape (4), puisque :

$$\forall i \in 1..n-1,\ \exists k \in 1..p,\ \exists j \in 1..n \text{ t.q. } \sigma(A_i) = \alpha_k(\sigma(X_j)) \text{ et } \sigma(B_i) = \beta_k(\sigma(Y_j))$$
$$\sigma(A_n) = \alpha_{j_n}(0) \text{ et } \sigma(B_n) = \beta_{j_n}(0)$$
$$\sigma(A_1) = \alpha_{j_1}(..\alpha_{j_r}(0)..) = \beta_{j_1}(..\beta_{j_r}(0)..) = \sigma(B_1)$$

□

Par conséquence, trouver une solution à PCP avec deux lettres n'est pas plus difficile que de trouver une exécution du protocole paramétré comme celui défini ci-dessus. En outre, trouver une exécution n'est pas plus difficile que de trouver une attaque où le secret peut être libéré à la fin de l'exécution. Ainsi, le problème d'insécurité des protocoles paramétrés sans aucune restriction est indécidable.

5.6.2 La classe des protocoles bien tagués avec clefs autonomes

Nous présentons dans cette section les restrictions adoptées en définissant la classe des protocoles bien tagués avec clefs autonomes pour laquelle nous prévoyons la décidabilité. Nous introduisons tout d'abord la notion d'autonomie :

Définition 5.6.2.1 *Autonomie.*

Un terme $mpair(i,u)$ est autonome quand $Var_{\mathcal{I}}(u) \subseteq \{i\}$. Quant à un terme $t \in T_{DY}$, il est autonome si $\#Var_{\mathcal{I}}(t) \leq 1$ et $\forall t' < t$, t' est autonome.

Un protocole $\mathcal{P} = \{R_\imath \Rightarrow S_\imath | \imath \in J\}$ est autonome ssi pour tout $\imath \in J$, $R_\imath$ et $S_\imath$ sont autonomes et $Var_{\mathcal{I}}(R_\imath) = \emptyset$ et $Var_{\mathcal{I}}(S_\imath) = \emptyset$.

Exemple 5.6.2.2 *Exemples de termes autonomes.*

Nous pouvons citer comme termes autonomes $mpair(i, \langle a_i, mpair(j, \{c_j\}_k)\rangle)$. Cependant, le terme $t = mpair(i, mpair(j, \{a_i\}_{c_j}))$ n'est pas autonome.

Nous introduisons maintenant la notion de protocoles bien tagués. L'idée de base derrière le taggage des variables est d'ajouter suffisamment d'informations pour les termes *mpair* afin d'éviter le fait que le protocole soit utilisé pour garantir des relations entre les éléments d'un même *mpair* tel que $\forall i,\ \exists i'$ t.q. $X_i = f(X_{i'})$. En effet, c'est sur ces entrelassements entre différents éléments de *mpair* que repose la preuve d'indécidabilité.

Définition 5.6.2.3 *Protocoles bien tagués.*

Un protocole $\mathcal{P} = \{R_\imath \Rightarrow S_\imath | \imath \in J\}$ est bien tagué ssi :

1. *$\forall \imath \in J$, $\forall X_i \in \mathcal{X}_{\mathcal{I}} \cap STermes(R_\imath) \cap \bigcup_{\imath' < \imath} STermes(R_{\imath'})$, X_i est taguée ;*
2. *$\forall \imath \in J$, $\forall X_i \in \mathcal{X}_{\mathcal{I}} \cap STermes(S_\imath)$, X_i est taguée ;*
3. *$\forall \imath \in J$, $\forall t = f(s_1,..,s_k) \in STermes(R_\imath \cup S_\imath)$ avec $f \neq mpair$, si $\exists j = 1..k$ t.q. s_j est tagué, alors t est tagué aussi avec le même tag ;*
4. *$\forall \imath \in J$, $\forall t \in STermes(R_\imath)$ tagué, $\forall X_i \leq t$ où $X_i \in \mathcal{X}_{\mathcal{I}}$, X_i est taguée.*
5. *$\mathcal{P}$ est autonome.*

Dans cette définition, les conditions 1 et 2 énoncent que toute variable indicée du protocole doit être taguée, à l'exception de sa première occurrence, modulo l'ordre *partiel* des étapes. Ensuite, la troisième condition (en combinaison avec les conditions 1 et 2) déclare que, pour chaque sous-terme t du protocole, si une variable indicée est accessible à partir de t par des

décompositions successives sans pour autant ouvrir aucun *mpair*, alors, ce terme t doit être tagué. Notons que, comme conséquence d'autonomie des *mpairs*, une variable indicée X_i peut apparaître soit taguée par son propre indice (comme dans ${X_i}^i$) ou bien non taguée. La condition 4 énonce que, toute variable indicée qui est sous-terme d'un terme tagué de R_i est taguée.

Exemple 5.6.2.4 *Termes et Protocoles bien tagués.*

Pour éclaircir la définition des protocoles bien tagués, nous donnons ici quelques exemples de ces protocoles qui sont parfois bien tagués et parfois ne le sont pas.

- *Soit $P = \{step_1, step_2\}$ avec, pour des termes t, t', une clef k et une variable indicée X_i, nous avons $step_1 = (mpair(i, X_i) \Rightarrow t)$ et $step_2 = (mpair(i, \{X_i\}_k) \Rightarrow t')$. Vu que la condition 1 n'est pas satisfaite pour P (X_i doit être taguée à l'étape $step_2$ alors que ce n'est pas le cas ici), ce protocole n'est pas bien tagué.*

- *Soit $(t \Rightarrow mpair(i, X_i))$ une étape du protocole P pour un terme t et une variable indicée X_i. Ainsi, P n'est pas un protocole bien taguée vu que la condition 2 n'est pas satisfaite puisque X_i doit être taguée alors qu'elle apparaît non taguée ici.*

- *Si $\left(mpair(i, \langle t', {X_i}^i\rangle) \Rightarrow t\right)$ ou $\left(t \Rightarrow mpair(i, \langle t', {X_i}^i\rangle)\right)$ constitue une étape du protocole P pour des termes t, t' et une variable indicée X_i, alors P n'est pas bien tagué. En effet, la condition 3 n'est pas satisfaite vu que $\langle t', {X_i}^i\rangle$ devrait être tagué.*

- *Soit $\left(mpair(i, \{X_i\}_k{}^i) \Rightarrow t\right)$ une étape d'un protocole P. P n'est pas alors bien tagué vu que la condition 4 n'est pas satisfaite puisque X_i devrait être taguée ici alors qu'elle ne l'est pas.*

- *Soit $P = \left(\{mpair(i, X_i) \Rightarrow mpair(i, \{{X_i}^i\}_k{}^i)\}\right)$ un protocole où X_i est une variable indicée et k est une clef. P est alors bien tagué car il vérifie toutes les conditions de la Définition 5.6.2.3.*

Nous revenons à notre étude de cas : le protocole de Asokan-Ginzboorg et nous donnons en Exemple 5.6.2.5 la version bien taguée de cette étude de cas.

Exemple 5.6.2.5 *Version bien taguée du protocole Asokan-Ginzboorg.*

$$
\begin{array}{ll}
(L,1) & Init \Rightarrow mpair(t, \langle l, \{e\}_p\rangle) \\
(S,1) & mpair(i, \langle L, \{E_i\}_p\rangle) \Rightarrow mpair(j, (\langle a_j, (\{\langle r_j, s_j\rangle\}_{(E_j)^j})^j\rangle)^j) \\
(L,2) & mpair(k, (\langle a_k, (\{\langle R_k, S_k\rangle\}_{(e)^k})^k\rangle)^k) \Rightarrow mpair(m, (\{\langle mpair(o, (S_o)^o), s'\rangle\}_{(R_m)^m})^m) \\
(S,2) & mpair(q, (\{\langle mpair(u, (s_u)^u), S'\rangle\}_{(r_q)^q})^q) \Rightarrow \\
 & mpair(w, \langle a_w, \{\langle (s_w)^w, H(\langle mpair(y, (s_y)^y), S'\rangle)\rangle\}_{F(\langle mpair(y,(s_y)^y), S'\rangle)}\rangle) \\
(L,3) & mpair(x, \langle a_x, \{\langle (S_x)^x, H(\langle mpair(z, (S_z)^z), s'\rangle)\rangle\}_{F(\langle mpair(z,(S_z)^z), s'\rangle)}\rangle) \Rightarrow End
\end{array}
$$

L'ordre partiel de ces étapes est le suivant : $W_L = 1, 2, 3$, $W_S = 1, 2$ avec $1 <_{W_L} 2 <_{W_L} 3$, et $1 <_{W_S} 2$.

Finalement, nous introduisons la notion de protocoles avec clefs autonomes. C'est pour cette classe de protocoles combinée avec celle des protocoles bien tagués que nous allons prouver la décidabilité en Chapitre 6. Cette notion (clefs autonomes) a été introduite pour prévenir le cas où les variables d'indices sont générées à partir des clefs, ce qui présente un argument fort pour la terminaison.

Définition 5.6.2.6 *Protocoles avec clefs autonomes.*

Un protocole $\mathcal{P} = \{R_\imath \Rightarrow S_\imath | \imath \in J\}$ *est dit avec clefs autonomes ssi* $\forall \imath \in J, \forall t \in STermes(R_\imath \cup S_\imath)$ *tel que* $t = \{u\}_v$, $Var_{\mathcal{I}}(v) = \emptyset$.

Exemple 5.6.2.7 *Exemples de termes avec (ou sans) clefs autonomes.*

Nous donnons ici quelques termes qui détiennent des clefs autonomes et d'autres sans cette caractéristique. Les termes $\{t\}_{s_i}$ *et* $\{t\}_{X_i}$ *n'ont pas de clefs autonomes. Cependant, les termes* $\{t\}_{mpair(i,s_i)}$ *et* $\{t\}_k$ *disposent des clefs autonomes.*

Nous donnons en Exemple 5.6.2.8 la version avec clefs autonomes de notre étude de cas : le protocole de Asokan-Ginzboorg

Exemple 5.6.2.8 *Version modifiée du protocole Asokan-Ginzboorg avec clefs autonomes.*
Dans cette nouvelle version, nous supposons que k est une clef symétrique, nommée généralement clef de chiffrement de clefs (KEK pour Key Encryption Key*) partagée entre le leader L et chaque participant* a_i *et donc le simulateur. La version avec clefs autonomes est alors spécifiée comme suit :*

$$\begin{array}{lllll} 1.\ S & \longrightarrow & L & : mpair(i, \langle a_i, \{s_i\}_k \rangle) \\ 2.\ L & \longrightarrow & S & : mpair(i, \{\langle mpair(j, s_j), s' \rangle\}_k) \\ 3.\ S & \longrightarrow & L & : mpair(i, \langle a_i, \{\langle s_i, h(\langle mpair(k, s_k), s' \rangle) \rangle\}_{f(\langle mpair(k,s_k),s' \rangle)} \rangle) \end{array}$$

5.7 Contraintes et système de contraintes

Nous utiliserons un système symbolique de contraintes pour représenter toutes les exécutions possibles d'un protocole étant donné un ordre *partiel* d'étapes.

Ce système utilise des quantificateurs (universels et existentiels) sur des variables d'indices tout en incluant un quantificateur universel implicite sur le nombre n d'éléments de tout $mpair$.

5.7.1 Préliminaires

Afin de définir notre système de contraintes, nous introduisons quelques notions de base telles que la notion de la relation entre sous-termes accessibles, ou encore celle d'environnement.

Pour deux termes s et s' (resp. deux ensembles de termes E et E'), nous notons $s \sim s'$ (resp. $E \sim E'$) s'ils sont égaux une fois les tags éliminés. Nous définissons par la suite la relation entre sous-termes accessibles.

Définition 5.7.1.1 *Relation* $\leqslant_E^L$ *pour sous-termes accessibles.*

Nous considérons la relation $\_ \leqslant \_$ *sur* $\mathcal{T} \times 2^{\mathcal{T}} \times 2^{\mathcal{T}} \times \mathcal{T}$. *Nous notons* $s \leqslant_E^L t$ *pour deux termes s et t dans* $\mathcal{T}$ *et E et L des sous-ensembles finis de* $\mathcal{T}$. *Notons que cette notation peut être aussi utilisée pour* $\mathcal{T}_s$.

Cette relation est définie comme étant la plus petite relation telle que :

$$\begin{array}{ll} t \leqslant_\emptyset^\emptyset t & \forall t \in \mathcal{T} \\ s \leqslant_E^L t & ssi\ \ s' \leqslant_{E'}^{L'} t' \ avec\ s \sim s', t \sim t', E \sim E', L \sim L' \\ Si\ \{m\}_k^p \leqslant_E^L t & alors\ \ m \leqslant_{E,k^{-1}}^{L'} t\ avec\ L' = L \cup \{\{m\}_k^p\} \\ Si\ \{m\}_b^s \leqslant_E^L t & alors\ \ m \leqslant_{E,b}^{L'} t\ avec\ L' = L \cup \{\{m\}_b^s\} \\ Si\ \langle t_1, \ldots, t_n \rangle \leqslant_E^L t & alors\ \ \forall i \leq n, t_i \leqslant_E^{L'} t\ avec\ L' = L \cup \{\langle t_1, \ldots, t_n \rangle\} \\ Si\ m \leqslant_{E'}^L t\ et\ E' \subset E & alors\ \ m \leqslant_E^L t \end{array}$$

Nous notons $u \leqslant_E t$ quand $u \leqslant_E^L t$ pour un certain ensemble L. Remarquons que, par construction, $u \leqslant_E t$ implique $u \in STermes(t)$. Nous disons dans ce cas que u est un sous-terme de t qui est accessible, i.e. il peut être obtenu par décompositions à partir de t en utilisant des clefs de E. L'ensemble L désigne l'ensemble de termes intermédiaires entre t et u y compris t. Pour une raison de simplicité, nous notons $\leqslant_{b_1,..,b_k}$ au lieu de $\leqslant_{\{b_1,..,b_k\}}$. Nous définissons aussi l'ensemble de termes strictement accessibles par $s <_F t$ (resp. $s <_F^L t$) si $s \leqslant_F t$ (resp. $s \leqslant_F^L t$) et $s \neq t$. Étant donné $t \leqslant_F^L u$ (resp $t <_F^L u$), nous appelons longueur de $t \leqslant_F^L u$ (resp $t <_F^L u$) le nombre d'éléments de L.

Exemple 5.7.1.2 *Exemples de relations $\leqslant_E^L$ et $<_E^L$.*

Nous donnons ici quelques petits exemples de la relation $\leqslant_E^L$ ainsi que de $\leqslant_E^L$. Comme exemple simple de $\leqslant_E^L$, nous pouvons citer $t \leqslant_\emptyset^\emptyset t$. Pour la relation $\leqslant_E^L$, nous pouvons mentionner : $t <_{\{k,k'\}}^{\{\{\langle t',\{t\}_{k'}\rangle\}_k,\langle t',\{t\}_{k'}\rangle,\{t\}_{k'}\}} \{\langle t',\{t\}_{k'}\rangle\}_k$. La longueur de ce dernier est le nombre d'éléments de $\{\{\langle t',\{t\}_{k'}\rangle\}_k,\langle t',\{t\}_{k'}\rangle,\{t\}_{k'}\}$ (3) qui représente le nombre de termes intermédiaires entre $\{\langle t',\{t\}_{k'}\rangle\}_k$ et t y compris $\{\langle t',\{t\}_{k'}\rangle\}_k$.

Nous introduisons maintenant ce que nous appelons environnement.

Définition 5.7.1.3 *Environnement.*

Nous appelons environment *un ensemble fini d'équations de type $(X = u)$ dont les parties gauches sont des variables ($X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$), précédé d'une liste de variables d'indices quantifiées existentiellement $\exists j_1, \ldots, j_p$ que nous notons $\exists R'$. Nous le notons habituellement $\mathcal{E}$.*

Intuitivement, l'environnement contient des équations représentatives des valeurs des variables. Lors de l'apparition d'une nouvelle équation pour une variable, l'équation représentative de cette même variable sera utilisée pour tester la cohérence entre les valeurs de cette même variable. Ainsi, les équations de l'environnement seront utilisées lors des entrelacements entre les contraintes d'une même variable quand nous voulons remplacer une variable par sa valeur et par la suite tester la cohérence des valeurs obtenues pour une même variable. Au début du traitement d'une étape d'une exécution, cet ensemble est mis à jour en ajoutant l'ensemble des équations représentatives des variables déduites lors du traitement de l'étape précédente de l'exécution en question.

5.7.2 Contraintes élémentaires et contraintes négatives

Après avoir introduit ces différentes notions, nous pouvons maintenant définir les différents types de contraintes ainsi que leurs solutions. L'ensemble de solutions d'une contrainte C, noté $[\![C]\!]_\tau^e$, où e est la valeur de n et τ est une substitution d'indices, est l'ensemble de substitutions closes qui seront définies ultérieurement dans cette section. Nous définissons $\mathcal{GS}$ comme l'ensemble de toutes les substitutions closes.

Nous commençons par les contraintes élémentaires qui sont définies dans la Définition 5.7.2.1.

Définition 5.7.2.1 *Contrainte élémentaire.*

Une contrainte élémentaire est une expression de type $(t \in Forge(E,\mathcal{K}))$, ou $(t = t')$, ou $(t \in Sub(t', E, \mathcal{E}, \mathcal{K}))$, ou $(t \in Sub_d(t', E, \mathcal{E}, \mathcal{K}))$ ou encore $(t \in Forge_c(E,\mathcal{K}))$ avec $t, t' \in \mathcal{T}$, $E \subset \mathcal{T}$ et $\mathcal{E}$ un environnement.

Une contrainte élémentaire représente des relations basiques entre les termes. En effet, la contrainte $t \in Forge(E,\mathcal{K})$ exprime le fait que le terme t est dérivable à partir de l'ensemble

des connaissances E sans utiliser aucun élément de l'ensemble $\mathcal{K}$ comme clef de déchiffrement. La contrainte $t \in Forge_c(E, \mathcal{K})$ a la même expression que $t \in Forge(E, \mathcal{K})$ sauf que, dans ce cas, t est obtenue par composition.

La contrainte $t \in Sub(t', E, \mathcal{E}, \mathcal{K})$ symbolise le fait que t est un sous-terme accessible de t' ayant E comme ensemble de connaissances, mais sans utiliser aucune clef des termes intermédiaires entre t et t' dans $\mathcal{K}$. Tout ceci est effectué modulo les remplacements utilisant les équations de l'environnement $\mathcal{E}$. La contrainte $t \in Sub_d(t', E, \mathcal{E}, \mathcal{K})$ admet la même expression que $t \in Sub(t', E, \mathcal{E}, \mathcal{K})$ sauf que cette fois, t est accessible à partir de t' seulement par décomposition. Finalement, la contrainte $t = t'$ exprime le fait que les deux termes t et t' sont égaux.

Plus formellement, l'ensemble de solutions des contraintes élémentaires est donné par la Définition 5.7.2.2.

Définition 5.7.2.2 *Solutions de contrainte élémentaire.*

$$
\begin{array}{lcl}
[\![\, t = t' \,]\!]^e_\tau & = & \{\sigma \in \mathcal{GS} \,|\, \overline{t}^e\tau\sigma = \overline{t'}^e\tau\sigma\} \\
[\![\, t \in Forge(E, \mathcal{K}) \,]\!]^e_\tau & = & \{\sigma \in \mathcal{GS} \,|\, \overline{t}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)\} \\
[\![\, t \in Forge_c(E, \mathcal{K}) \,]\!]^e_\tau & = & \{\sigma \in \mathcal{GS} \,|\, \overline{t}^e\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)\}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\, t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau & = & \{\sigma \in \mathcal{GS} \,|\, \exists u\ \exists F, L \text{ t.q. } u \leqslant^L_F \overline{w}^e\tau,\ F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma), \\
& & F\sigma \cap \overline{\mathcal{K}}^e\tau\sigma = \emptyset,\ \textit{et soit } u\sigma = \overline{t}^e\tau\sigma,\ \textit{soit } \exists v, \delta, k,\ \tau' \text{ t.q.} \\
& & \left\{ \begin{array}{l}
\left\{ \begin{array}{l}
\textit{soit } u \in \mathcal{X},\ (u = v) \in \mathcal{E},\ \delta = \emptyset,\ \textit{et } \tau' = \tau \\
\textit{soit } \exists \overrightarrow{Z} \in \overrightarrow{\mathcal{X}},\ i, j \in \mathcal{I} \text{ t.q. } u = Z_i\tau',\ (Z_j = v) \in \mathcal{E}, \\
\qquad \delta = \delta^k_{j,i}, \tau \subseteq \tau' \textit{ et } \mathcal{D}om(\tau') = \mathcal{D}om(\tau) \cup \{k, i\}
\end{array} \right. \\
k, i \notin Var_{\mathcal{I}}(\{t, w, \mathcal{E}\}), u\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma) \\
u\sigma = \overline{v}^e\delta\tau'\sigma,\ \textit{et } \sigma \in [\![\, t \in Sub(v\delta, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_{\tau'} \qquad \}
\end{array} \right.
\end{array}
$$

$[\![\, t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ est définie d'une manière similaire à $[\![\, t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ sauf que pour le premier cas (quand $u\sigma = \overline{t}^e\tau\sigma$), nous avons $u <^L_F \overline{w}^e\tau$.

Dans cette définition, l'ensemble de solutions d'une contrainte de type $t \in Forge(E, \mathcal{K})$, étant donné une substitution d'indices τ et un entier e, est l'ensemble de substitutions closes σ tel que $\overline{t}^e\tau\sigma$ soit dérivable à partir de l'ensemble $\overline{E}^e\tau\sigma$ sans pour autant utiliser un élément de $\overline{\mathcal{K}}^e\tau\sigma$ comme clef de déchiffrement.

Pour un ensemble de clefs $\mathcal{K}$ et un ensemble de connaissances E, nous disons qu'un terme u est accessible à partir d'un terme v si u est accessible de v tout en générant les clefs intermédiaires entre ces deux termes à partir de $\overline{E}^e\tau\sigma$ sans l'aide d'un terme de $\overline{\mathcal{K}}^e\tau\sigma$. L'ensemble de solutions pour $t \in Sub(w, E, \mathcal{E}, \mathcal{K})$, étant donné une substitution d'indices τ et un entier e, est l'ensemble de substitutions closes σ tel que :

- soit $\overline{t}^e\tau\sigma$ est un terme accessible de $\overline{w}^e\tau\sigma$ (pour l'ensemble des clefs $\overline{\mathcal{K}}^e\tau\sigma$ et l'ensemble des connaissances $\overline{E}^e\tau\sigma$),
- soit il existe une variable (indicée ou non-indicée) sous-terme accessible de w telle que, par remplacement de cette variable en utilisant une équation qui lui est représentative appartenant à l'environnement $\mathcal{E}$, notre terme $\overline{t}^e\tau\sigma$ devient un terme accessible de la valeur donnée par cette équation choisie.

Exemple 5.7.2.3 *Exemples d'ensembles de solutions de contraintes élémentaires.*

Nous utilisons la notation $[X \leftarrow c]$ pour définir une substitution qui associe la valeur c à la variable X. Nous suivons toujours les mêmes conventions d'écriture de termes que celles

présentées en Section 5.4. Les termes indicés désignent soit des variables indicées (en majuscules) soit des constantes indicées (minuscules).

$$[\![\{X\}_k = \{c_i\}_Y]\!]^e_{\tau,i\leftarrow 1} = \{[X \leftarrow c_1, Y \leftarrow k]\}$$

$$[\![f(X) \in Forge(\{\{f(c)\}_k, k\}, \{k\})]\!]^e_\tau = \emptyset$$

$$[\![\{f(X_i)\}_k \in Forge_c(\{f(c), k, \{f(r_j)\}_k\}, \emptyset)]\!]^e_{\tau,i\leftarrow 1,j\leftarrow 2} = \{[X_1 \leftarrow c]\}$$

$$[\![\{X\}_k \in Sub(\{\langle c, \{r_i\}_k\rangle\}_k, \emptyset, \emptyset, \emptyset)]\!]^e_{\tau,i\leftarrow 1} = \{[X \leftarrow \langle c, \{r_i\}_k\rangle]\}$$

$$[\![\{X\}_k \in Sub(\{\langle c, \{r_i\}_k\rangle\}_k, \{k', \{k\}_{k'}\}, \emptyset, \{k'\})]\!]^e_{\tau,i\leftarrow 1} = \{[X \leftarrow \langle c, \{r_i\}_k\rangle]\}$$

$$[\![\{X\}_k \in Sub_d(\{\langle c, \{r_i\}_k\rangle\}_k, \{k', \{k\}_{k'}\}, \emptyset, \{k'\})]\!]^e_{\tau,i\leftarrow 1} = \emptyset$$

$$[\![\{X\}_k \in Sub(\{\langle c, \{r_i\}_k\rangle\}_k, \{k', \{k\}_{k'}\}, \emptyset, \emptyset)]\!]^e_{\tau,i\leftarrow 1} = \{[X \leftarrow \langle c, \{r_i\}_k\rangle] \cup [X \leftarrow r_1]\}$$

$$[\![\{X\}_k \in Sub_d(\{\langle c, \{r_i\}_k\rangle\}_k, \{k', \{k\}_{k'}\}, \emptyset, \emptyset)]\!]^e_{\tau,i\leftarrow 1} = \{[X \leftarrow r_1]\}$$

$$[\![\{X\}_k \in Sub_d(\{\langle c, Y\rangle\}_k, \{k\}, \{(Y = \{r\}_k)\}, \emptyset)]\!]^e_\tau = \{[X \leftarrow r]\}$$

$$[\![\{X\}_k \in Sub_d(\{\langle c, Y_i\rangle\}_k, \{k\}, \{(Y_j = \{r_j\}_k)\}, \emptyset)]\!]^e_{\tau,i\leftarrow 1} =$$
$$[\![\{X\}_k \in Sub_d(\{\langle c, \{r_i\}_k\rangle\}_k, \{k\}, \{(Y_j = \{r_j\}_k)\}, \emptyset)]\!]^e_{\tau,i\leftarrow 1} = \{[X \leftarrow r_1]\}$$

Soit τ' *tel que* $\tau, i \leftarrow 1 \subset \tau'$ *et* $\tau'(l) = 2$ *pour* $l \notin \mathcal{D}om(\tau)$:
$$[\![\{X\}_k \in Sub_d(\{\langle c, Y_i\rangle\}_k, \{k\}, \{(Y_j = \{r_m\}_k)\}, \emptyset)]\!]^e_{\tau,i\leftarrow 1} =$$
$$[\![\{X\}_k \in Sub_d(\{\langle c, \{r_l\}_k\rangle\}_k, \{k\}, \{(Y_j = \{r_m\}_k)\}, \emptyset)]\!]^e_{\tau'} = \{[X \leftarrow r_2]\}$$

Nous définissons maintenant les contraintes négatives.

Définition 5.7.2.4 *Contrainte négative.*
Une contrainte négative est une expression de la forme $(\forall i\ X_m \neq u)$ *ou* $(X_m \notin Forge_c(E, \mathcal{K}))$ *avec* $X_m \in \mathcal{X}_\mathcal{I}$, $u \in \mathcal{T}$, $E, E' \subset \mathcal{T}$ *et* $i \in Var_\mathcal{I}(u)$.

La première contrainte a pour but d'éliminer parmi les solutions celle(s) qui donnent à la variable X_m la valeur u pour une valeur quelconque de la variable d'indice i de u. La deuxième contrainte sert essentiellement à éliminer des solutions qui donnent à la variable X_m une valeur constructible à partir d'un ensemble de connaissances E'. Plus formellement, l'ensemble de solutions de ces contraintes, étant donné une substitution d'indice τ et un entier e est décrit dans la définition suivante.

Définition 5.7.2.5 *Solutions de contrainte négative.*

$$\begin{aligned} [\![X_m \notin Forge_c(E, \mathcal{K})]\!]^e_\tau &= \mathcal{GS} \setminus [\![X_m \in Forge_c(E, \mathcal{K})]\!]^e_\tau \\ [\![(\forall i\ X_m \neq u)]\!]^e_\tau &= \mathcal{GS} \setminus \bigcup_{x=1\ldots e} [\![X_m = u]\!]^e_{[i\leftarrow x],\tau} \end{aligned}$$

5.7.3 Blocs de contraintes et système de contraintes

Nous décrivons maintenant notre système de contraintes qui est représenté par des blocs.

Définition 5.7.3.1 *Système de contraintes.*

Tout d'abord, nous définissons un bloc de contraintes B comme étant la conjonction de contraintes pour un environnement $\mathcal{E}$:

$$B = (ctr_1 \wedge \ldots \wedge ctr_l, \mathcal{E})$$

Pour simplifier les notations, nous allons considérer les blocs comme des ensembles de contraintes élémentaires ou négatives. Par exemple, nous écrivons $c \in B$ pour exprimer le fait qu'une contrainte élémentaire c est un élément de la conjonction de contraintes de B.

Nous pouvons maintenant définir le système de contraintes qui va être utilisé pour représenter les exécutions du protocole. Étant données deux listes finies de variables d'indices $Q = i_1, \ldots, i_k$ et $R = j_1, \ldots, j_l$, nous écrivons le préfixe de quantificateurs $\forall i_1 \cdots \forall i_k \exists j_1 \cdots \exists j_l$, en bref : $\forall Q\ \exists R$.

Un système de contraintes, noté S, est une disjonction de blocs avec un préfixe de quantificateurs :

$$S \quad = \quad \forall Q\ \exists R\ (B_1 \vee \ldots \vee B_p)$$

L'idée est d'utiliser ce système de contraintes basé sur les blocs pour représenter toutes les manières possibles à l'intrus pour construire une liste de termes représentée par un *mpair*. Nous aurons un bloc du système pour chacune de ces manières.

Nous définissons maintenant l'ensemble de solutions d'un système de contraintes comme suit :

Définition 5.7.3.2 *Solutions d'un système de contraintes.*

Considérons un système de contraintes S, B_i, pour $i = 1 \ldots p$ (blocs de S) et $ctr_{i,j}$, pour $j = 1 \ldots l_i$ (contraintes du bloc B_i) donné par la Définition 5.7.3.1. L'ensemble de solutions d'un système de contraintes est défini inductivement par différents cas :

$$\begin{array}{lcllcl}
[\![\forall i\ S]\!]^e_\tau & = & \cap_{x=1,\ldots,e} [\![S]\!]^e_{[i \leftarrow x],\tau} & [\![B_1 \vee .. \vee B_p]\!]^e_\tau & = & \bigcup\limits_{i=1\ldots p} [\![B_i]\!]^e_\tau \\
[\![\exists i\ S]\!]^e_\tau & = & \cup_{x=1,\ldots,e} [\![S]\!]^e_{[i \leftarrow x],\tau} & [\![B_i]\!]^e_\tau & = & \bigcap\limits_{j=1\ldots l_i} [\![ctr_{i,j}]\!]^e_\tau
\end{array}$$

D'après cette définition, l'ensemble de solutions d'un système de contraintes (sans quantificateurs), modulo une substitution d'indice, est l'ensemble de solutions qui satisfait au moins l'un des blocs de ce système de contraintes. D'une manière similaire, l'ensemble de solutions d'un bloc de contraintes (sans quantificateurs), modulo une substitution d'indice, est l'ensemble de solutions qui satisfait toutes les contraintes formant ce bloc. Ensuite, l'ensemble de solutions d'un système de contraintes avec un quantificateur universel est l'ensemble de solutions satisfaisant ce système sans ce quantificateur pour toutes les valeurs que peut prendre la variable d'indice du quantificateur. De la même manière, l'ensemble de solutions d'un système de contraintes avec un quantificateur existentiel est l'ensemble de solutions satisfaisant ce système sans ce quantificateur pour au moins une valeur parmi les valeurs que peut prendre la variable d'indice du quantificateur.

Exemple 5.7.3.3 *Exemples de solutions de systèmes de contraintes.*

$$\begin{aligned}
&[\![\forall i \ ((X_i = c) \wedge (X_j = r))\,]\!]^e_{\tau,j\leftarrow 1} \\
&= \cap_{x=1,\ldots,e} [\![\,((X_i = c) \wedge (X_j = r))\,]\!]^e_{\tau,j\leftarrow 1,i\leftarrow x} \\
&= [\![\,(X_j = r)\,]\!]^e_{\tau,j\leftarrow 1,i\leftarrow x} \cap \cap_{x=1,\ldots,e} [\![\,(X_i = c)\,]\!]^e_{\tau,j\leftarrow 1,i\leftarrow x} \\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
&[\![\exists i \ ((X_i = c) \wedge (X_j = r))\,]\!]^e_{\tau,j\leftarrow 1} \\
&= \cup_{x=1,\ldots,e} [\![\,((X_i = c) \wedge (X_j = r))\,]\!]^e_{\tau,j\leftarrow 1,i\leftarrow x} \\
&= [X_1 \leftarrow r, X_2 \leftarrow c] \cup \ldots \cup [X_1 \leftarrow r, X_e \leftarrow c]
\end{aligned}$$

Ensuite, les blocs sont étendus pour admettre aussi des contraintes étiquetées :

Notation 5.7.3.4 *Étiquettes des contraintes.*

Une contrainte ctr peut être équipée d'une étiquette $(ctr)^m$ *ou* $(ctr)^{sm}$ *ou* $(ctr)^f$ *ou encore* $(ctr)^d$ *pour désigner respectivement une contrainte maître ou sous-maître ou finale ou encore endormie. Les deux premières étiquettes permettent de maintenir une valeur* représentative *d'une variable indicée ou non-indicée. Par exemple, nous montrerons dans le Chapitre 6, que nous avons exactement une contrainte maître pour toute variable indicée dans chaque bloc. Ces contraintes maîtres et sous-maîtres seront utilisées pour instancier des variables quand c'est nécessaire. La troisième étiquette sera employée pour des contraintes afin d'empêcher toute autre réécriture sur ces contraintes. La quatrième étiquette sera utilisée elle aussi pour des contraintes afin d'empêcher toute autre réécriture sur ces contraintes à moins que l'ensemble des contraintes maîtres change soit par addition d'une nouvelle contrainte maître, soit par modification d'une existante.*

Nous introduisons aussi la notation $(ctr)^\bullet$ *qui désigne une contrainte étiquetée ou non étiquetée. Les solutions des contraintes étiquetées sont les solutions des mêmes contraintes sans étiquettes.*

Pour simplifier l'utilisation des contraintes maîtres ou sous-maîtres, nous les groupons en deux ensembles :

Définition 5.7.3.5 *Ensemble de contraintes (sous-)maîtres.*

Soit $S = \forall Q \exists R\ B_1 \vee .. \vee B_p$ *un système de contraintes,* $Y \in \mathcal{X}$, $W \subseteq \mathcal{X}$ *et* $\overrightarrow{X} \in \overrightarrow{\mathcal{X}}$. *Alors,*

$$\mathcal{M}(S, \overrightarrow{X}) = \{ctr \,|\, \exists i\ (ctr)^m \in B_i \text{ et } \exists j \in Q \text{ t.q } ctr = (X_j \in Forge_c(E, \mathcal{K})) \text{ ou } ctr = (X_j = u)\}$$

$$\mathcal{SM}(B_i, W) = \{ctr \,|\, (ctr)^{sm} \in B_i,\ Y \in W \text{ et } ctr = (Y = u)\}\,.$$

$$\mathcal{SM}(B_i, Y) = \mathcal{SM}(B_i, \{Y\}).$$

Quand S est clairement défini par le contexte, nous l'omettons de $\mathcal{M}(S, \overrightarrow{X})$ en le notant simplement $\mathcal{M}(\overrightarrow{X})$.

5.8 Système de règles d'inférence

Nous avons défini dans la Section 5.7 notre système de contraintes. Ce système doit être mis à jour lors des traitements des différentes étapes d'une exécution du protocole.

Nous présentons dans cette section les règles de la fonction de normalisation qui seront appliquées à ce système de contraintes selon trois phases (voir la définition de la fonction de

normalisation en Définition 6.2.0.5). Ces règles sont organisées en six groupes selon leurs objectifs (voir Section 5.8.2). Les règles du premier groupe sont prioritaires sur les autres groupes puisqu'elles assurent le contrôle des formes des contraintes. Après chaque application d'une de ces règles d'inférence au système de contraintes, le résultat est mis en forme normale disjonctive et les quantificateurs existentiels sont mis en préfixe du système en utilisant la logique du premier ordre. L'ensemble de toutes ces règles d'inférence forment un système de transformation qui garantit l'équivalence entre systèmes de contraintes.

5.8.1 Quelques notions nécessaires

Nous introduisons tout d'abord quelques notions nécessaires pour la définition des règles telles que la notion d'*historique de bloc* (Définition 5.8.1.1), l'antériorité des variables indicées d'un même bloc (Définition 5.8.1.2), la fonction *awake* (Définition 5.8.1.3) et le graphe de dépendance (Définition 5.8.1.4).

Nous commençons tout d'abord par étendre les blocs de contraintes afin d'inclure un historique, à savoir, une liste ordonnée de règles qui ont été appliquées à une certaine étape dans ce bloc :

Définition 5.8.1.1 *Historique de bloc.*

Chaque bloc $B = ctr_1 \wedge .. \wedge ctr_p$ *est équipé d'une liste ordonnée de règles* $[r_1, \ldots r_k]$ *nommée historique du bloc* B*, notée* $Hist(B)$*.* $Hist(B)(i)$ *désigne la* $i^{\text{ème}}$ *règle de l'historique. Le bloc est alors désigné par* $B = (ctr_1 \wedge .. \wedge ctr_p, \mathcal{E}, Hist)$*, et les systèmes de contraintes sont étendus à des blocs avec historiques.*

Toute application de règles mettra à jour automatiquement l'historique du bloc auquel cette règle est appliquée. Par conséquent, dans ce qui suit, nous aurons toujours l'historique d'une manière implicite et nous ne l'écrirons pas s'il n'est pas approprié pour la discussion.

Nous introduisons aussi la notion d'*antériorité* entre variables d'indices dans le même bloc qui sera utilisée dans la définition de la Règle [5.25] et dans les preuves. Pour cela, pour une règle d'inférence r, nous désignons par $\text{post}(r)$ la partie droite de r.

Définition 5.8.1.2 *Antériorité des indices.*

Soit $B = (ctr_1 \wedge .. \wedge ctr_p, \mathcal{E}, Hist)$ *un bloc. Soient* i *et* j *deux variables d'indices.*
Nous définissons la fonction $ant(,)$ *de* $\mathcal{I} \times \mathcal{I}$ *dans* $\mathcal{I}$ *telle que*
$ant(i,j) = i$ *si* $\exists \imath$ *t.q.* $i \in Var_{\mathcal{I}}(post(Hist(B)(\imath)))$ *et* $\forall \imath' \leq \imath\ j \notin Var_{\mathcal{I}}(post(Hist(B)(\imath')))$.

Nous définissons ensuite la fonction *awake* qui a pour but d'enlever les étiquettes pour les contraintes endormies. Ceci est utilisé quand une contrainte maître est introduite dans le système ou une contrainte maître de type $Forge$ vient d'être changée en une nouvelle contrainte maître de type $\acute{E}galit\acute{e}$.

Définition 5.8.1.3 *La fonction awake.*

Nous définissons la fonction awake comme suit :

$$awake((ctr)^d) = ctr$$
$$awake((ctr)^\bullet) = (ctr)^\bullet \text{ pour } (ctr)^\bullet \neq (ctr)^d$$

La fonction awake est étendue d'une manière standard pour les blocs de contraintes et les systèmes de contraintes. En effet, soit $B = ctr_1 \wedge \ldots \wedge ctr_m$ *un bloc de contraintes et* $S =$

$B_1 \vee \ldots \vee B_l$ *un système de contraintes. Alors,*

$$awake(B) = awake(ctr_1) \wedge \ldots \wedge awake(ctr_m)$$
$$awake(S) = awake(B_1) \vee \ldots \vee awake(B_l)$$

Finalement, nous définissons le graphe de dépendance d'un bloc, qui sera utilisé pour définir la notion d'accessibilité de variables, utilisée pour la Règle [5.3].

Définition 5.8.1.4 *Graphe de dépendance d'un bloc.*
Nous définissons le graphe de dépendance $\mathcal{G}_B$ d'un bloc B comme étant le graphe où $\mathcal{X} \cup \mathcal{X}_\mathcal{I}$ est l'ensemble de noeuds et nous avons un arc orienté de X à Y ssi il existe $(X = u) \in B$ avec $Y < u$. Nous notons $X \sqsubset Y$ quand il existe un chemin dans $\mathcal{G}_B$ de X à Y. Nous notons aussi $X \sqsubset_l Y$ si $X \sqsubset Y$ et l est la longueur minimale du chemin de X à Y.

5.8.2 Les règles d'inférence

Nous pouvons maintenant présenter notre système de règles. Ces règles sont organisées en six groupes $G_1, \ldots, G_6$ selon leur but. En résumé, G_1 vise à maintenir des propriétés syntaxiques du système de contraintes. G_2 contient les règles de $Forge$ qui analysent les différentes manières de construire un terme (par composition ou décomposition des connaissances). Ces règles énumèrent donc les différentes manières de composer un terme par l'intrus. G_3 contient les règles de Sub qui sont similaires à celles de $Forge$, sauf que celles-là sont employées pour décomposer les connaissances de l'intrus. G_4 code la résolution des contraintes d'égalité comme un algorithme d'unification pour les termes. G_5 a pour but de remplacer les variables par leur valeur, dans un même bloc, indépendamment des autres blocs. Finalement, G_6 généralise G_5 en permettant les remplacements des variables d'un bloc à un autre. Tous ces groupes seront détaillés dans les paragraphes qui suivent.

G_1 : Groupe des règles prioritaires
Les règles de ce groupe permettent de garder des propriétés de syntaxe ou de formattage concernant le système de contraintes. Toutes les règles de ce groupe doivent être appliquées (si elles peuvent l'être) avant les règles des autres groupes. En outre, les règles sont données par l'ordre de priorité strictement décroissant. Par exemple, la Règle [5.1] est prioritaire sur la Règle [5.2].

$$t = X \quad \longrightarrow \quad X = t \text{ où } X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I} \text{ et } t \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I} \qquad [5.1]$$

$$(X = u)^{sm} \wedge (Y = X) \quad \longrightarrow \quad (X = u)^{sm} \wedge (Y = u) \text{ pour } X, Y \in \mathcal{X} \qquad [5.2]$$

$$X = u \quad \longrightarrow \quad \bot \text{ s'il existe } Y < u \text{ t.q. } Y \sqsubset X \text{ pour } X, Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I} \qquad [5.3]$$

$$\begin{aligned} \forall Q.i\ \exists R\ S \vee (ctr \wedge B) \quad &\longrightarrow \quad \forall Q.i\ \exists R\ awake(S) \vee ((ctr)^m \wedge awake(B)) \\ \text{où} \quad ctr \in \ &\{X_i \in Forge_c(E, \mathcal{K}), X_i = u\} \text{ et } B \cap \mathcal{M}(\overrightarrow{X}) = \emptyset \end{aligned} \qquad [5.4]$$

$$\begin{array}{rcl} \forall Q.i.j\ \exists R\ S \vee ((X_i \in Forge_c(E,\mathcal{K}))^m \wedge X_j = u \wedge B) & \longrightarrow & \forall Q.i.j\ \exists R \\ awake(S) \vee (X_i \in Forge_c(E,\mathcal{K}) \wedge (X_j = u)^m \quad \wedge awake(B)) & & \end{array} \quad [5.5]$$

$$(Y_j = Z)^{\bullet} \wedge (X = u) \quad \longrightarrow \quad (Y_j = Z)^{\bullet} \wedge (X = u\lambda) \text{ où } Y_j < u \text{ et } \lambda = [Y_j \leftarrow Z] \quad [5.6]$$

$$\begin{array}{rcl} B \wedge (X = u) & \longrightarrow & B \wedge (X = u\lambda) \wedge (Y_j = Z) \text{ où } Y_j < u, \text{ et } \lambda = [Y_j \leftarrow Z] \\ & & \text{si pour tout } Z' \text{ nous avons } (Y_j = Z') \notin B \end{array} \quad [5.7]$$

$$B \wedge (X = u) \quad \longrightarrow \quad B \wedge (X = u)^{sm} \text{ où } \mathcal{SM}(B, X) = \emptyset \quad [5.8]$$

Parmi ces règles du groupe G_1, il existe des règles qui manipulent les étiquettes des contraintes maîtres en ajoutant de nouvelles étiquettes ou bien en transférant des étiquettes existantes. En effet, la Règle [5.4] marque une contrainte comme étant maître pour un vecteur de variables $\overrightarrow{X} \in \overrightarrow{\mathcal{X}}$ dans un bloc B quand B ne contient pas déjà une contrainte maître pour $\overrightarrow{X}$. Notons que la Règle [5.4] est appliquée uniquement une fois par vecteur de variables dans un bloc. La Règle [5.5] transfère des étiquettes des contraintes maîtres de type $Forge$ à des contraintes de type égalité quand c'est possible. Intuitivement, une contrainte d'égalité est beaucoup plus précise qu'une contrainte de type $Forge$ et par la suite, elle doit être favorisée. Notons ici que l'indice en lui même, i.e. i ou j dans la Règle [5.5], n'est pas approprié au choix des contraintes (maître et celle qui devrait l'être). Notons aussi que, pour ces deux règles, puisqu'elles modifient l'ensemble de contraintes maîtres, les contraintes endormies du bloc qui contient la contrainte maître (nouvelle ou modifiée) et du système en général, doivent être réveillées. Ceci est assuré par l'application de la fonction *awake*.

La Règle [5.2] présente un cas particulier de remplacements de variables pour des contraintes d'égalité mais cette fois, concernant les variables constituant la partie droite de ces contraintes et uniquement lorsqu'il s'agit d'une égalité de type variable égale à une autre variable.

La Règle [5.3] permet d'éliminer le bloc qui présente des contraintes pouvant mener à des cycles dans ce même bloc. Nous utilisons le graphe de dépendance pour détecter ces cycles (par exemple, $X = f(.., X, ..)$).

D'autres règles formattent les contraintes afin d'avoir, de préférence, des variables dans la partie gauche de ces contraintes comme pour la Règle [5.1], ou bien remplacent des variables indicées par d'autres non-indicées comme pour les Règles [5.6] et [5.7].

Finalement, la Règle [5.8] manipule les contraintes sous-maîtres en s'assurant que, si un certain bloc contient au moins une égalité fixant une valeur à une variable X, alors il existe exactement une contrainte parmi celles-ci qui est étiquetée sous-maître pour X.

G_2 : Groupe des règles de réduction de Forge

G_2 permet d'énumérer toutes les différentes possibilités de construire un terme par l'intrus. Nous considérons dans ce paragraphe un bloc de contraintes $(B', \mathcal{E})$.

$$t \in Forge(E,\mathcal{K}) \quad \longrightarrow \quad t \in Forge_c(E,\mathcal{K}) \vee \bigvee_{w \in E} t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \quad [5.9]$$

$$\langle t_1, \ldots, t_m \rangle \in Forge_c(E, \mathcal{K}) \longrightarrow \bigwedge_{i=1\ldots m} t_i \in Forge(E, \mathcal{K}) \quad [5.10]$$

$$\{t\}_b \in Forge_c(E, \mathcal{K}) \longrightarrow b \in Forge(E, \mathcal{K}) \wedge t \in Forge(E, \mathcal{K}) \quad [5.11]$$

$$h(t) \in Forge_c(E, \mathcal{K}) \longrightarrow t \in Forge(E, \mathcal{K})) \quad [5.12]$$

$$\begin{array}{rll} \forall Q\ \exists R\ S \vee (B \wedge mpair(k,t) \in Forge_c(E,\mathcal{K})) & \longrightarrow & \\ \forall Q\ \exists R\ S \vee (B \wedge t\delta \in Forge(E,\mathcal{K})) & \text{si} & Hy\ [5.13] \\ \forall Q.k'\ \exists R\ S \vee (B \wedge t\delta_{k,k'} \in Forge(E,\mathcal{K})) & & \text{sinon avec } k' \text{ frais} \\ \text{où } Hy\ [5.13] \text{ est } ((mpair(k,t) \in Forge_c(E,\mathcal{K}) & \longrightarrow & t\delta \in Forge(E,\mathcal{K})) \in Hist(B')) \\ \text{avec } B' = (B \wedge mpair(k,t) \in Forge_c(E,\mathcal{K})) & & \end{array} \quad [5.13]$$

$$c \in Forge_c(E, \mathcal{K}) \longrightarrow \bot, \text{ pour } c \in \mathcal{C} \cup \mathcal{C}_{\mathcal{I}} \quad [5.14]$$

La Règle [5.9] est une règle générique qui illustre les deux possibilités pour construire un terme t : soit par sa composition à partir de ses composants (sous-termes) ou bien par décomposition de l'une des connaissances (t est donc un sous-terme déductible).

Le reste des règles énumère toutes les possibilités pour que l'intrus puisse composer un terme : nous avons exactement une règle pour décomposer chaque type d'opérateurs de la signature $\mathcal{G}$ (Règles [5.10] à [5.13]). En particulier, la Règle [5.10] compose une paire de termes, la Règle [5.11] compose un chiffrement et la Règle [5.12] compose un hachage. La Règle [5.13] opère pour l'opérateur *mpair* qui modélise une liste : pour composer une liste, il faut composer tous ses éléments. D'où l'introduction de la variable d'indice k quantifiée universellement. En outre, cet indice k est frais (noté k' dans la règle) si la contrainte initiale n'a jamais été traitée par la Règle [5.13] auparavant et donc l'application de cette règle à cette contrainte n'a pas été enregistrée dans l'historique du bloc contenant la contrainte. Dans le cas contraire, i.e. cette contrainte a été traitée auparavant alors nous utilisons le même indice qui a été généré auparavant. Ceci présente un argument parmi d'autres pour prouver l'existence d'une borne pour l'ensemble des indices générés par notre système de contrainte. Notons aussi que l'autonomie des *mpair* est utilisée ici pour justifier la place du quantificateur universel k à l'intérieur du préfixe des quantificateurs. Finalement, le Règle [5.14] traite le cas de base lorsqu'il s'agit d'une constante : on ne peut pas composer une constante.

G_3 : Groupe des règles de réduction de Sub

Les règles de G_3 sont similaires à celles de G_2 à la différence qu'elles décomposent les connaissances de l'intrus.

$$\begin{array}{rll} t \in Sub(u, E, \mathcal{E}, \mathcal{K}) & \longrightarrow & (t = u) \text{ si } u = \{v\}_b \text{ et } b \in \mathcal{K} \\ & & (t = u) \vee (t \in Sub_d(u, E, \mathcal{E}, \mathcal{K})) \text{ sinon} \end{array} \quad [5.15]$$

$$t \in Sub_d(\langle t_1, \ldots, t_m \rangle, E, \mathcal{E}, \mathcal{K}) \longrightarrow \bigvee_{i=1\ldots m} (t \in Sub(t_i, E, \mathcal{E}, \mathcal{K})) \quad [5.16]$$

$$t \in Sub_d(\{u\}_b^s, E, \mathcal{E}, \mathcal{K}) \longrightarrow t \in Sub(u, E, \mathcal{E}, \mathcal{K}) \wedge b \in Forge(E, \mathcal{K} \cup \{b\}) \quad [5.17]$$

$$t \in Sub_d(\{u\}_K^p, E, \mathcal{E}, \mathcal{K}) \longrightarrow t \in Sub(u, E, \mathcal{E}, \mathcal{K}) \wedge K^{-1} \in Forge(E, \mathcal{K} \cup \{K\}) \quad [5.18]$$

$$\begin{array}{rll} t \in Sub_d(mpair(k,u), E, \mathcal{E}, \mathcal{K}) & \longrightarrow & \quad [5.19] \\ (t \in Sub(u\delta, E, \mathcal{E}, \mathcal{K})) & & \text{si } Hy[5.19] \\ \exists k' \; (t \in Sub(u\delta_{k,k'}, E, \mathcal{E}, \mathcal{K})) & & \text{sinon} \end{array}$$

où k' frais et $Hy[5.19]$ est $((t \in Sub_d(mpair(k,u), E, \mathcal{E}, \mathcal{K}) \longrightarrow t \in Sub(u\delta, E, \mathcal{E}, \mathcal{K})) \in Hist(B))$

$$t \in Sub_d(c, E, \mathcal{E}, \mathcal{K}) \longrightarrow \bot, \text{ pour } c \in \mathcal{C} \cup \mathcal{C}_{\mathcal{I}} \quad [5.20]$$

La Règle [5.15] est une règle générique qui suit précisément les règles de déduction de l'intrus : un terme t est un sous-terme accessible de u ssi $t = u$ ou il existe un sous-terme direct u' de u, dérivable de u, et tel que t est un sous-terme accessible de u'. Ainsi, il existe exactement une règle pour chaque type d'opérateur de $\mathcal{G}$ (mis à part les variables et les constantes).

En particulier, la Règle [5.16] décompose une paire de termes, les Règles [5.17] et [5.18] décomposent un chiffrement. La Règle [5.19] opère pour l'opérateur *mpair*. Cette règle illustre le fait que pour construire un terme t à partir d'une décomposition d'un *mpair* qui représente une liste, il suffit de construire t à partir d'un élément de cette liste et par la suite il existe un élément de la liste, à partir duquel, nous pouvons déduire t. D'où l'introduction d'une variable d'indice k quantifiée existentiellement. En outre, cet indice k est frais (noté k' dans la règle) si la contrainte initiale n'a jamais été traitée par la Règle [5.19] auparavant et donc l'application de cette règle à cette contrainte n'a pas été enregistré dans l'historique du bloc contenant la contrainte. Dans le cas contraire, i.e. cette contrainte a été traitée auparavant alors nous utilisons l'indice déjà généré. Finalement, le Règle [5.20] traite le cas de base lorsqu'il s'agit d'une constante pour exprimer qu'on ne peut pas décomposer une constante.

G_4 : Groupe des règles de simplification des égalités

Les règles de G_4 codent l'algorithme d'unification pour les termes de notre système, et par la suite, elles permettent la résolution des contraintes d'égalité.
Notons $\widehat{t}$ le symbole de la racine du terme t. Par exemple, $\widehat{f(t)} = f$.

$$c = c \longrightarrow \top \text{ où } c \in \mathcal{C} \cup \mathcal{C}_{\mathcal{I}} \quad [5.21]$$

$$t = t' \longrightarrow \bot \text{ où } \{\widehat{t}, \widehat{t'}\} \subset \mathcal{C} \cup \mathcal{C}_{\mathcal{I}} \cup \mathcal{G} \text{ et } \widehat{t} \neq \widehat{t'} \quad [5.22]$$

$$f(u_1,..,u_m) = f(w_1,..,w_m) \quad \longrightarrow \quad \bigwedge_{i=1...m} u_i = w_i \ \text{ où } f \in \{\{\}^p, \{\}^s, \langle\rangle, [], h\} \qquad [5.23]$$

$$\begin{array}{rll} \forall Q\ \exists R\ S \vee (B \wedge (mpair(k,u) = mpair(l,w))) & \longrightarrow & \qquad [5.24] \\ \forall Q\ \exists R\ S \vee (B \wedge (u\delta = w\delta_{l,k}\delta)) & & si\ Hy[5.24] \\ \forall Q.k'\ \exists R\ S \vee (B \wedge (u\delta_{k,k'} = w\delta_{l,k'})) & & \text{sinon} \\ \text{où } k' \text{ frais et } Hy[5.24] \text{ est } (((mpair(k,u) = mpair(l,w)) & \longrightarrow & (u\delta = w\delta_{l,k}\delta)) \in Hist(B')) \\ \text{avec } B' = (B \wedge (mpair(k,u) = mpair(l,w))) & & \end{array}$$

$$\begin{array}{rll} \forall Q\ \exists R\ S \vee (B \wedge c_j = c_i) & \longrightarrow & \forall Q\ \exists R\ S \vee (B\delta) \ \text{ où } c_i, c_j \in \mathcal{C}_{\mathcal{I}} \qquad [5.25] \\ & & \text{et } \delta = \delta_{i,j} \text{ si } j \in Q \\ & & \text{et } \delta = \delta_{j,i} \text{ si } i \in Q \\ & & \text{et } \delta = \delta_{\{i,j\},ant(i,j)} \text{ sinon} \end{array}$$

Ces règles, et plus particulièrement les Règles [5.21] et [5.22] , consistent simplement à tester récursivement la compatibilité des deux opérateurs se trouvant à la racine des deux termes dont on veut tester l'égalité. Notons que ces règles décrivent également l'égalité entre termes tagués. En effet, deux termes tagués sont égaux si leurs tags sont égaux et les termes sans tags sont égaux, ce qui est assuré par l'ajout à notre signature de l'opérateur spécial de taggage [_].

La Règle [5.24] teste si deux termes *mpair* sont égaux. Deux listes sont égales si leurs éléments sont deux à deux égaux en respectant l'ordre de ces listes, d'où la nécessité d'un seul quantificateur universel pour comparer les deux listes élément par élément. La variable d'indice quantifiée universellement peut être fraîche (le cas de k') si la contrainte considérée n'a pas été traitée auparavant dans le bloc en question (le cas où l'hypothèse [5.24] n'est pas satisfaite). Dans le cas contraire, cette contrainte a été traitée dans ce bloc et donc l'application de la Règle [5.24] à cette contrainte a été enregistré dans l'historique de ce bloc : nous réutilisons donc dans la contrainte résultante l'indice généré lors du premier traitement de cette contrainte (ici k). Ceci évite une explosion du nombre d'indices et permet de prouver l'existence d'une borne pour le nombre d'indices générés dans notre système de contraintes.

La Règle [5.25] teste l'égalité entre deux constantes indicées du même vecteurs de constantes. Ces deux constantes sont égales si leurs indices sont égaux, d'où le besoin de faire des remplacements d'indices que l'on effectue de la manière suivante :

- nous remplaçons l'indice quantifié existentiellement par celui quantifié universellement,
- dans le cas où les deux indices sont quantifiés existentiellement, nous gardons l'indice antérieur, celui qui est apparu le premier dans l'historique du bloc. Ce choix sert à aider à borner l'ensemble des indices générés dans notre système de contraintes.

Par conséquent, seules les contraintes d'égalité restent après itération de ces règles ; elles associent une valeur à une variable, i.e. $X = u$ avec $X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$.

G_5 : Groupe des règles d'entrelacements dans un même bloc

Les règles de G_5 ont pour but de remplacer les variables par leurs valeurs dans un même bloc. Elles manipulent les interactions entre les contraintes dans le même bloc, indépendamment

des autres blocs. Elles considèrent alors les occurrences multiples d'une même variable, avec le même indice pour le cas de variables indicées.

$$\begin{array}{rcl} B \wedge (X_i = u)^{\bullet} \wedge (X_i = v)^{\bullet} & \longrightarrow & B \wedge (X_i = u)^{\bullet} \wedge (X_i = v)^{\bullet} \wedge u = v \\ \text{si } (B \wedge (X_i = u)^{\bullet} \wedge (X_i = v)^{\bullet} & \longrightarrow & B \wedge (X_i = u)^{\bullet} \wedge (X_i = v)^{\bullet} \wedge u = v) \notin Hist(B') \\ \text{avec } B' = B \wedge (X_i = u)^{\bullet} \wedge (X_i = v)^{\bullet} & & \end{array} \quad [5.26]$$

$$(X = u)^{sm} \wedge (X = v) \longrightarrow (X = u)^{sm} \wedge (u = v) \quad [5.27]$$

$$(X_i = u)^{\bullet} \wedge X_i \in Forge_c(E', \mathcal{K}) \longrightarrow (X_i = u)^{\bullet} \wedge u \in Forge_c(E', \mathcal{K}) \quad [5.28]$$

$$(X = u)^{sm} \wedge X \in Forge_c(E', \mathcal{K}) \longrightarrow (X = u)^{sm} \wedge u \in Forge_c(E', \mathcal{K}) \quad [5.29]$$

$$\begin{array}{rcl} (A \in Forge_c(E, \mathcal{K}))^{\bullet} \wedge A \in Forge_c(E', \mathcal{K}) & \longrightarrow & (A \in Forge_c(E, \mathcal{K}))^{\bullet} \\ & \text{où} & E \subseteq E' \text{ et } A \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}} \end{array} \quad [5.30]$$

$$\begin{array}{rcl} (X = w)^{sm} \wedge t \in Sub_d(X, E', \mathcal{E}, \mathcal{K}) & \longrightarrow & (X = w)^{sm} \wedge t \in Sub_d(w, E', \mathcal{E}, \mathcal{K}) \\ \wedge X \notin Forge_c(E, \mathcal{K}) & \text{où} & (X = w)^{sm} \in \mathcal{E} \end{array} \quad [5.31]$$

$$A \in Forge_c(E, \mathcal{K}) \wedge t \in Sub_d(A, E', \mathcal{E}, \mathcal{K}) \longrightarrow \bot \text{ où } A \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}} \quad [5.32]$$

La Règle [5.26] teste la cohérence de deux valeurs assignées à une même variable indicée en ajoutant au bloc la contrainte d'égalité de ces deux valeurs. Notons que les contraintes d'égalité concernant la variable indicée peuvent être non taguées ou taguées même avec les tags endormis ou finaux. La seule condition d'arrêt de l'application de cette règle est de l'avoir déjà appliquée à ces mêmes contraintes. Ceci est testé par l'appartenance de cette règle avec les mêmes contraintes à l'historique du bloc.

La Règle [5.27] transforme une contrainte d'égalité, qui assigne une valeur à une variable non indicée, en une autre contrainte d'égalité testant la cohérence de cette valeur avec la valeur donnée par la contrainte sous-maître de cette même variable. Notons que l'intérêt des contraintes sous-maîtres est de les utiliser pour tester la cohérence d'autres éventuelles valeurs de ces variables dans un bloc donné.

Les Règles [5.28] et [5.29] traitent l'entrelacement entre deux contraintes dans un même bloc, concernant la même variable (indicée ou non-indicée), l'une de type $Forge$ et l'autre de type égalité. Puisque, pour une variable donnée, la contrainte d'égalité est plus précise que celle de type $Forge$, nous gardons la contrainte d'égalité. Cependant, nous remplaçons la contrainte $Forge$ par une autre contrainte $Forge$ qui aura pour but de tester la construction de la valeur de la variable en question à partir des informations données par la contrainte $Forge$ initiale.

La Règle [5.30] exprime le fait que, si l'intrus est capable de composer une variable à partir d'un petit ensemble de connaissances, alors il peut la composer à partir d'un ensemble plus grand.

La Règle [5.32] exprime le fait qu'il n'est pas nécessaire de décomposer une variable qu'on a composé. En effet, toute information (sous-terme) que l'intrus pourrait avoir de la décomposition d'une variable, il a pu l'obtenir directement puisqu'il a pu composer cette variable et donc ses sous-termes.

Notons que, bien que la sémantique de $Sub_d(\_,\_,\_)$ était un peu plus complexe que prévu, elle a été définie pour empêcher des actions inutiles comme ceci, et par la suite assurer la validité des règles de G_6.

Cette sémantique intervient aussi explicitement dans la Règle [5.31]. En effet, cette règle remplace une variable à l'intérieur d'un Sub par sa valeur donnée par la contrainte sous-maître de cette variable extraite de l'environnement $\mathcal{E}$. Elle ajoute aussi une contrainte $(X \notin Forge_c(E, \mathcal{K}))$ pour exprimer le fait que, puisque l'intrus a réussi à décomposer cette variable, il est inutile de composer à nouveau cette dernière.

G_6 : Groupe des règles d'entrelacement entre différents blocs

Les règles de G_6 généralisent celles de G_5 en permettant les remplacements des variables d'un bloc à un autre. Par conséquent, ces règles définissent les interactions entre les blocs et résolvent des contraintes concernant une même variable mais avec des indices multiples d'une manière formelle. Étant donné un vecteur de variables $\overrightarrow{X} \in \overrightarrow{\mathcal{X}}$, nous supposons que

$$\mathcal{M}(\overrightarrow{X}) = \{X_{i_o} = u_o\}_{o=1..p} \cup \{X_{j_r} \in Forge_c(E'_r, K_r)\}_{r=1..q}.$$

Ainsi, les règles de réécriture de ce groupe pour des contraintes concernant $\overrightarrow{X}$ sont le suivantes, avec $\delta_o = \delta^{k'_o}_{Q,m}$ et $k'_o \in \mathcal{I}$ est une variable d'indice fraîche :

$$t \in Sub_d(X_m, E', \mathcal{E}, \mathcal{K}) \longrightarrow \bigvee_{(X_i = u) \in \mathcal{E}} \exists k'\ t \in Sub_d(u\delta, E', \mathcal{E}, \mathcal{K}) \quad [5.33]$$
$$\wedge\, (X_m = u\delta)^f \wedge X_m \notin Forge_c(E', \mathcal{K})$$
$$\text{où } \delta = \delta^{k'}_{i,m}$$

$$X_m \in Forge_c(E', \mathcal{K}) \longrightarrow ((X_m \in Forge_c(E', \mathcal{K}))^d \quad [5.34]$$
$$\wedge \bigwedge_{o=1...p} \left(\forall k'_0\ X_m \neq u_o\delta_o\right))$$
$$\vee \bigvee_{o=1...p} (\exists k'_0\ u_o\delta_o \in Forge_c(E', \mathcal{K}) \wedge (X_m = u_o\delta_o)^f)$$

$$X_m = v \longrightarrow \bigvee_{r=1...q} ((v \in Forge_c(E'_r, \mathcal{K}_r)) \wedge (X_m = v)^d \wedge \quad [5.35]$$
$$\bigwedge_{o=1...p} \left(\forall k'_0\ X_m \neq u_o\delta_o\right))$$
$$\vee \bigvee_{o=1...p} (\exists k'_o\ (u_o\delta_o = v) \wedge (X_m = u_o\delta_o)^f)$$

La structure de ces règles est essentiellement la même que pour l'entrelacement dans un seul bloc : étant donnée une contrainte concernant une variable indicée X_i qui doit être remplacée par sa valeur, nous énumérons un nombre fini de termes *candidats* pour X_i et qui représentent donc toutes les valeurs possibles de X_i selon tout le système de contraintes. Ces valeurs sont fournies par les contraintes maîtres.

Par exemple, dans la Règle [5.33], seul le cas d'entrelacement avec des contraintes maîtres de type égalité est pris en compte puisque l'entrelacement avec celles de type $Forge_c$ mène à $\perp$. Ceci est dû au fait qu'il est inutile de décomposer une variable que l'intrus a déjà composé. Cette règle ajoute une contrainte d'égalité représentant la contrainte maître utilisée, en la taguant finale afin d'empêcher toute autre réécriture de cette contrainte. Elle ajoute aussi une contrainte négative pour exprimer le fait que, puisque l'intrus a réussi à décomposer cette variable, il est inutile de composer à nouveau cette dernière. Il est possible aussi que cette règle génère des variables d'indices susceptibles d'être fraîches. Cette génération d'indices est contrôlée selon le paragraphe suivant.

D'une manière similaire, la Règle [5.34] remplace une variable dans une contrainte $Forge_c$ par sa valeur selon le type des contraintes maîtres utilisées. Dans le cas d'utilisation d'une contrainte de type $Forge_c$, la contrainte initiale est taguée comme endormie en attendant la modification de l'ensemble de contraintes maîtres pour pouvoir la réveiller. Ensuite, pour distinguer la valeur de la variable de toute valeur provenant des autres contraintes maîtres de type égalité, une conjonction de contraintes négatives est introduite au bloc. Dans le cas inverse, i.e. utilisation des contraintes maîtres d'égalité, nous testons la possibilité de construction de la valeur choisie par la contrainte maître modulo remplacement d'indices. La contrainte maître utilisée est transférée, modulo remplacement d'indices, au bloc en question et elle est taguée finale pour empêcher toute éventuelle réécriture.

Finalement, dans la Règle [5.35], l'entrelacement concerne une contrainte d'égalité. Comme pour les autres règles de ce même groupe, nous utilisons l'ensemble de contraintes maîtres et nous distinguons l'entrelacement avec des contraintes maîtres de type $Forge_c$ de celui avec des contraintes maîtres d'égalité. Si l'entrelacement est effectuée avec une contrainte maître de type $Forge_c$, alors nous testons si la valeur de la variable donnée par la contrainte initiale peut être composée à partir des informations données par la contrainte maître. Ensuite, la contrainte initiale est endormie. Finalement, pour assumer le choix de type de la contrainte maître, et donc la distinguer des autres valeurs données par les contraintes maîtres de type égalité, une conjonction de contraintes négatives est ajoutée au bloc. Si, par contre, l'entrelacement est fait avec une contrainte maître de type égalité, alors, pour chaque contrainte maître choisie, nous testons la cohérence entre les deux valeurs de la variable tout en transférant cette contrainte maître en la taguant comme finale.

Contrôle de génération d'indices pour G_6

Pour contrôler la génération d'indices par les règles du groupe G_6, nous avons préféré le faire hors de la définition de ces règles pour simplifier la présentation. Nous définissons donc dans cette section ces contrôles effectués pour ces règles. Soit B le bloc sur lequel nous nous focalisons lors de l'application des Règles [5.33], [5.34] et [5.35].

Pour la Règle [5.33], si $(t \in Sub_d(X_m, E, \mathcal{E}, \mathcal{K}) \longrightarrow \exists k'\ t \in Sub_d(u\delta, E', \mathcal{E}, \mathcal{K}) \wedge (X_m = u\delta)^f \wedge X_m \notin Forge_c(E', \mathcal{K}))$ appartient à $Hist(B)$, alors nous sauvegardons le même indice k' pour $u\delta$. C'est-à-dire, nous générons la même contrainte Sub qui a été produite auparavant et qui a été enregistré dans $Hist(B)$. Dans le cas contraire, i.e. la contrainte $(t \in Sub_d(X_m, E, \mathcal{E}, \mathcal{K}))$ n'a pas déjà été traitée avec la Règle [5.33] par un entrelacement avec une contrainte maître

$(X_i = u)$, alors k' sera un indice frais.

Le même raisonnement est valide pour la Règle [5.34]. En effet, si nous n'avons pas déjà traité la contrainte $(X_m \in Forge_c(E', \mathcal{K}))$ par la Règle [5.34] par entrelacement avec une contrainte maître $(X_{i_o} = u_o)$, alors l'indice k'_o est frais. Sinon, nous préservons le même indice qui a été généré auparavant par la Règle [5.34] pour la contrainte $(X_m \in Forge_c(E', \mathcal{K}))$ en considérant une contrainte maître $(X_{i_o} = u_o)$ et qui appartenait à $Hist(B)$.

Nous raisonnons d'une manière similaire pour la Règle [5.35]. En effet, si la contrainte $(X_m = v)$ a été déjà traité par la Règle [5.35] par l'entrelacement avec une contrainte maître $(X_{i_o} = u_o)$ et donc $(X_m = v \longrightarrow \exists k'_o\ (u_o\delta_o = v) \wedge (X_m = u_o\delta_o)^f)$ appartient à $Hist(B)$, alors nous préservons la même variable d'indice qui a été générée auparavant. Sinon, k'_o sera une variable d'indice fraîche.

5.8.3 Les règles d'inférence et le taggage

Nous avons défini notre système de règles d'inférence pour des termes non tagués. Cependant, ce système se base sur des contraintes manipulant des termes susceptibles d'être tagués. Nous rappelons que la notion de taggage a été introduite pour éviter d'avoir des relations entre les éléments d'une liste représentée par un *mpair*. En effet, une telle éventuelle relation est une source d'indécidabilité comme le montre la Section 5.6. Notre système de règles a donc intérêt à assurer le traitement de ce genre de termes.

Les règles d'inférence définies dans la Section 5.8.2 sont aussi applicables pour des termes tagués en suivant la définition de notre signature. En effet, le but de ce qui suit dans cette section est de détailler le traitement des termes tagués par nos différents groupes de règles.

Tout d'abord, les règles de réduction des contraintes $Forge$ et Sub sont aussi valables pour traiter les termes tagués. En effet, les règles génériques des groupes G_1 et G_2, à savoir les Règles [5.9] et [5.15], sont exactement pareilles pour les termes tagués. Par exemple, la Règle [5.9] pour un terme tagué $(t)^i$ est :

$$(t)^i \in Forge(E, \mathcal{K}) \longrightarrow (t)^i \in Forge_c(E, \mathcal{K}) \vee \bigvee_{w \in E} (t)^i \in Sub(w, E, \mathcal{E}, \mathcal{K})$$

Puis, pour les autres règles de décomposition que ce soit pour les contraintes $Forge_c$ ou celles de Sub, traiter les termes tagués est exactement similaire au traitement des termes sans tags. Par exemple, la Règle [5.10] devient pour le traitement d'un terme $(\langle t_1, \ldots, t_m \rangle)^j$:

$$(\langle t_1, \ldots, t_m \rangle)^j \in Forge_c(E, \mathcal{K}) \longrightarrow \bigwedge_{i=1\ldots m} t_i \in Forge(E, \mathcal{K})$$

Ensuite, pour les contraintes d'égalité, la contrainte $(X_i)^i = u$ mène à $\perp$ si jamais u est non-tagué, puisque $(X_i)^i = [e_i, X_i]$. De la même manière, $(X_i)^i = (u)^j$ déduit que les deux tags i et j sont égaux.

En outre, pour les remplacements dans des contraintes, les variables taguées se comportent comme des variables *spéciales*. En effet, lors de ces remplacements, nous remplaçons la variable en question par sa valeur tout en conservant son tag. Le remplacement d'une variable de la forme $(X_i)^i$ par une valeur (un terme) u a alors la forme de $(u)^i$, pour le simple fait que nous remplaçons la variable X_i au sein du terme $[e_i, X_i]$. Par exemple, étant donné une contrainte $(t \in Sub({X_i}^i, E, \mathcal{E}, \mathcal{K}))$, nous cherchons des contraintes maîtres pour le vecteur de variables $\overrightarrow{X}$.

Puis, supposons que $(X_o = u)^m$ est une contrainte parmi ces contraintes maîtres, le résultat du remplacement serait $(t \in Sub((u\delta)^i, E, \mathcal{E}, \mathcal{K}))$. Ce même raisonnement est valide pour les contraintes $Forge_c$.

5.8.4 Contrainte en forme normale

Le but de l'application de toutes les règles définies dans la Section 5.8.2 est d'arriver à un système de contraintes dites en *forme normale*, i.e. sur lesquelles aucune règle n'est applicable. Le but d'avoir un tel système est d'en tester la satisfaisabilité afin de trouver une solution décrivant une attaque. Nous donnons dans cette section la définition de contrainte en forme normale qui sera utilisée dans le Chapitre 6. Nous appliquons ensuite nos règles d'inférence sur l'exemple du protocole Asokan-Ginzboorg en présentant les contraintes en forme normale trouvées.

Définition 5.8.4.1 *Contrainte en forme normale.*

Une contrainte en forme normale est une contrainte de type : $(X_i \in Forge_c(E, \mathcal{K}))^\bullet$, *ou* $X \in Forge_c(E, \mathcal{K})$, *ou* $(X_i = u)^\bullet$, *ou* $(X = u)^{sm}$, *ou* $(\forall j\ X_i \neq u)$ *ou encore* $(Y \notin Forge_c(E, \mathcal{K}))$, *avec* $X \in \mathcal{X}$, $Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, $X_i \in \mathcal{X}_\mathcal{I}$, $u \in \mathcal{T}$, $j \in Var_\mathcal{I}(u)$, $E \subset \mathcal{T}$ *et* $\mathcal{K} \subset \mathcal{T}$

Nous montrerons dans le Chapitre 6 que à chaque étape d'une exécution du protocole, le système de contraintes normalisées (obtenues quand aucune règle n'est applicable) contient uniquement des contraintes en forme normale.

Exemple 5.8.4.2 *Application au protocole de Asokan-Ginzboorg.*

Considérons la spécification du protocole de Asokan-Ginzboorg donnée par l'Exemple 5.4.2.1. Nous nous intéressons uniquement à l'étape $(S, 1)$ *de cette spécification :*

$$mpair(i, \langle L, \{E_i\}_p\rangle) \in Forge(E_1, \mathcal{K}_1).$$

Supposons que $E_1 = \{mpair(t, \langle l, \{e\}_p\rangle)\}$ *et* $\mathcal{E} = \emptyset$.

$mpair(i, \langle L, \{E_i\}_p\rangle) \in Forge(E_1, \emptyset)$
$\longrightarrow \quad (mpair(i, \langle L, \{E_i\}_p\rangle) \in Forge_c(E_1, \emptyset))$
$\quad\quad \vee (mpair(i, \langle L, \{E_i\}_p\rangle) \in Sub(mpair(t, \langle l, \{e\}_p\rangle), E_1, \mathcal{E}, \emptyset))$
par application de la Règle [5.9]

$\longrightarrow \quad \forall i\ ((\langle L, \{E_i\}_p\rangle \in Forge(E_1, \emptyset)) \vee (mpair(i, \langle L, \{E_i\}_p\rangle) = mpair(t, \langle l, \{e\}_p\rangle)))$
$\quad\quad \vee (mpair(i, \langle L, \{E_i\}_p\rangle) \in Sub_d(mpair(t, \langle l, \{e\}_p\rangle), E_1, \mathcal{E}, \emptyset)))$
par application des Règles [5.13] *et* [5.15]

$\longrightarrow \quad \forall i\ \forall j\ ((\langle L, \{E_i\}_p\rangle \in Sub(mpair(t, \langle l, \{e\}_p\rangle), E_1, \mathcal{E}, \emptyset)) \vee (\langle L, \{E_j\}_p\rangle = \langle l, \{e\}_p\rangle)$
$\quad\quad \vee (mpair(i, \langle L, \{E_i\}_p\rangle) \in Sub_d(\langle l, \{e\}_p\rangle, E_1, \mathcal{E}, \emptyset)) \vee (\langle L, \{E_i\}_p\rangle \in Forge_c(E_1, \emptyset)))$
par application des Règles [5.9], [5.19], [5.15], [5.22] *et* [5.24]

$\longrightarrow \quad \forall i\ \forall j\ ((L \in Forge(E_1, \emptyset) \wedge \{E_i\}_p \in Forge(E_1, \emptyset)) \vee ((L = l)^{sm} \wedge E_j = e)$
$\quad\quad \vee (\langle L, \{E_i\}_p\rangle \in Sub(\langle l, \{e\}_p\rangle, E_1, \mathcal{E}, \emptyset)))$
par application des Règles [5.10], [5.15], [5.22], [5.19], [5.16], [5.20] *et* [5.23]

$\longrightarrow \quad \forall i \, \forall j \, ((L \in Forge(E_1, \emptyset) \wedge E_i \in Forge(E_1, \emptyset) \wedge p \in Forge(E_1, \{E_i\}_p))$
$\vee \, (L \in Forge(E_1, \emptyset) \wedge \{E_i\}_p \in Sub_d(mpair(t, \langle l, \{e\}_p \rangle), E_1, \mathcal{E}, \emptyset))$
$\vee \, ((L = l)^{sm} \wedge E_i = e) \vee ((L = l)^{sm} \wedge E_j = e))$
par application des Règles [5.9], [5.11], [5.15], [5.22], [5.23], [5.16], [5.17] *et* [5.20]

$\longrightarrow \quad \forall i \, \forall j \, ((L \in Forge(E_1, \emptyset) \wedge E_i = e) \vee ((L = l)^{sm} \wedge E_i = e) \vee ((L = l)^{sm} \wedge E_j = e))$
par application des Règles [5.15], [5.19], [5.16], [5.17], [5.20], [5.22] *et* [5.23]

$\longrightarrow \quad \forall i \, \forall j \, ((L \in Forge(E_1, \emptyset) \wedge (E_i = e)^m) \vee ((L = l)^{sm} \wedge (E_i = e)^m)$
$\vee \, ((L = l)^{sm} \wedge (E_j = e)^m))$
par application de la Règle [5.4]

5.9 Conclusion

Nous avons proposé dans ce chapitre une approche pour la vérification de protocoles de groupe, caractérisés par le nombre de participants non borné qu'ils impliquent, et plus généralement pour la vérification de protocoles manipulants des listes paramétrées. L'intuition derrière cette approche est de considérer une classe de protocoles de groupe où il existe deux types de participants : un leader (ou serveur) et un certain nombre de participants *ordinaires* qui ont des comportements similaires vis-à-vis de la réception et de l'envoi des messages. Nous modélisons en conséquence tous les participants *ordinaires* en un seul participant *spécial* appelé *simulateur*.

Nous avons introduit une transformation pour les protocoles de groupe qui permet de passer du modèle asynchrone au modèle synchrone d'un protocole. Certes cette transformation n'est pas toujours complète, i.e. nous pouvons perdre d'éventuelles attaques, mais si nous trouvons une attaque dans notre modèle synchrone alors il s'agit d'une vraie attaque. Nous avons aussi précisé les conditions de l'équivalence entre ces deux modèles, sous couvert d'une pré-analyse du protocole.

Il est à noter que, mis à part celui des protocoles de groupe, ce modèle peut être aussi utilisé dans d'autres domaines tels que les services Web où les messages peuvent contenir des listes non bornées de nœuds XML encryptés. Dans ce cas, aucun changement du modèle n'est nécessaire.

Nous avons donc présenté notre modèle synchrone pour les protocoles de groupe qui généralise les modèles classiques tels que [89] en incluant des listes non bornées dans les messages. Pour pouvoir gérer ces listes, un nouvel opérateur a été introduit. Il est noté *mpair* et désigne une liste construite sur une racine unique (*pattern*). L'ajout naïf de cet opérateur mène à l'indécidabilité du problème de l'insécurité, ce qui nous a conduit à introduire une classe de protocoles appelée la classe des protocoles *bien tagués* avec *clefs autonomes*, qui est décidable.

Nous avons exposé ensuite les différentes contraintes que nous utilisons, pour enfin détailler nos règles d'inférence portant sur des systèmes de contraintes (i.e. la représentation formelle d'un ensemble infini d'états du protocole). Ces différentes règles d'inférence permettent de définir la procédure de vérification de la sécurité pour la classe des protocoles bien tagués avec clefs autonomes que nous introduirons dans le prochain chapitre (Chapitre 6). Nous montrerons que le système de règles, sur lequel se base cette procédure, est désormais correct et complet. Nous prouverons que l'application de ces règles pour un système de contraintes termine et que la forme normale obtenue peut être testée pour la satisfaisabilité. Ainsi, nous présenterons dans le chapitre prochain une procédure de décision pour cette classe.

6

Décidabilité pour les protocoles paramétrés bien tagués à clefs autonomes

Sommaire

6.1 Introduction

Nous avons proposé dans le chapitre précédent un modèle synchrone pour les protocoles de groupe qui peut être aussi utilisé pour tout autre protocole utilisant des listes paramétrées par leur longueur. La gestion de ces listes a été assurée par l'ajout d'un opérateur appelé *mpair* qui modélise une liste de termes ayant une forme commune (le même *pattern*). Cependant, le

problème de l'insécurité pour ce modèle avec cet opérateur s'avère indécidable. Nous avons donc introduit une classe de ces protocoles appelée la classe des protocoles bien tagués avec clefs autonomes. Nous avons ensuite présenté le système de contraintes que nous considérons et qui sera manipulé par l'ensemble de règles d'inférence que nous avons proposé.

Ce chapitre constitue une suite du précédent, puisque son objectif est de proposer une procédure de décision pour la classe de protocoles introduite dans ce dernier, i.e. la classe des protocoles bien tagués avec clefs autonomes.

Nous commençons donc par définir en Section 6.2 la procédure de vérification considérée qui se base essentiellement sur l'application des règles introduites dans le chapitre précédent, et qui résout le problème de l'insécurité des protocoles. Nous énoncerons dans la même section les résultats obtenus pour cette procédure, et en particulier le résultat de décidabilité pour la classe considérée.

Dans la Section 6.3, nous revenons sur quelques définitions présentées dans le chapitre précédent et utilisées dans les différentes preuves des sections qui suivent. Nous prouverons par la suite en Section 6.4 que l'ensemble de nos règles sont correctes et complètes. Nous justifierons en Section 6.5 que notre modèle est effectivement une extension des modèles classiques des protocoles cryptographiques en prouvant que, en l'absence d'indices ou de *mpair* pouvant générer des indices, nos règles d'inférence terminent. Dans la Section 6.6, nous montrerons la terminaison pour la classe des protocoles bien tagués avec clefs autonomes. Nous prouverons aussi en Section 6.7 que la normalisation d'un système de contraintes par notre ensemble de règles conduit à des contraintes en forme normale dont on peut tester la satisfaisabilité. Nous obtiendrons ainsi un résultat de décidabilité pour la classe considérée, celle des protocoles bien tagués avec clefs autonomes. Nous donnons dans la Section 6.8 des exemples testés par application de notre procédure.

6.2 Vérification des protocoles bien tagués avec clefs autonomes

Nous introduisons dans cette section une procédure de vérification de protocoles cryptographiques qui se base essentiellement sur le système de règles d'inférence définies dans la Section 5.8.2 du Chapitre 5. Nous énoncerons ensuite le résultat de décidabilité pour la classe de protocoles bien tagués avec clefs autonomes. Ce résultat est le fruit de différents résultats intermédiaires qui seront aussi énoncés dans cette section. Nous justifierons aussi le fait que notre modèle est une extension des modèles classiques de vérification de protocoles grâce à un résultat de terminaison pour les protocoles sans indices et sans *mpair* pouvant générer ces indices.

Étant donné un ensemble R de règles d'inférence et une formule F, $R(F)$ est une *clôture* de F par R si elle est dérivée par un nombre fini d'applications des règles de R et aucune règle ne peut être appliquée à $R(F)$. Avant de présenter notre procédure, nous introduisons quelques définitions telles que celle de la clôture de l'environnement utilisé dans l'algorithme de vérification définie par la Définition 6.2.0.3, la notion de niveau de variable définie en Définition 6.2.0.4, ou encore la définition de la fonction de normalisation (Définition 6.2.0.5).

Nous commençons tout d'abord par la définition d'une réduction de chaîne d'égalités au sein de l'environnement $\mathcal{E}$, étant donné un ensemble de variables d'indices quantifiées universellement Q. L'idée de cette clôture de l'environnement par des règles est d'utiliser une valeur (un terme) au lieu d'une variable lors du remplacement d'une autre variable. Cette clôture est définie comme suit :

Notation 6.2.0.3 $\lceil \mathcal{E} \rceil$.

Nous notons $\lceil \mathcal{E} \rceil$ la clôture de l'environnement $\mathcal{E}$ par les règles suivantes :

$$\begin{array}{rcl} X = Y \wedge Y = u & \longrightarrow & X = u \wedge Y = u \\ X_i = Y_i \wedge Y_j = u & \longrightarrow & \exists k' \, X_i = u\delta_{Q,i}^{k'} \wedge Y_j = u \end{array}$$

pour $X, Y \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$, $X_i, Y_j \in \mathcal{X}_{\mathcal{I}}$, $u \notin \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$ et k' frais.

Ensuite, nous avons besoin d'une autre notion appelée *niveau de variable*. Cette notion est essentielle pour déceler les propriétés de notre système de contraintes qui seront utilisées ensuite pour les preuves de correction et complétude.

Cette notion de niveau de variable et autres est donnée par la Définition 6.2.0.4.

Définition 6.2.0.4 *Niveau de variable, niveau de vecteur de variable.*

Supposons que P, Sec et S_0 sont comme dans la Définition 5.4.4.2 et π un ordre d'exécution correct pour le protocole, nous notons $E_i = S_0, S_1, .., S_i$ pour tout $i \in 1..k$. Soit $A \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$. Alors, $\mathcal{L}(A)$ est le plus petit ensemble E_i pour $i = 1 \ldots k$ tel que $A \leq R_{i+1}$. Nous étendons la notion de niveau au vecteur de variable de la manière suivante : Soit $\overrightarrow{X} \in \overrightarrow{\mathcal{X}}$. Alors, $\mathcal{L}(\overrightarrow{X})$ est le plus petit ensemble E_i pour $i = 1 \ldots k$ tel que $\exists m \in \mathcal{I}$ avec $E_i = \mathcal{L}(X_m)$.

Ensuite, nous définissons la fonction de normalisation d'un système de contraintes S, noté par $S \mapsto (S)\downarrow$. La normalisation d'un système de contraintes peut être définie comme étant les clôtures successives du système de contraintes en utilisant différents ensembles de règles parmi les règles d'inférence définies dans la Section 5.8.2 du Chapitre 5. Cette normalisation procède par deux phases principales séparées par une troisième phase d'étiquetage :

Définition 6.2.0.5 *La fonction de normalisation.*

Soit S un système de contraintes sous forme de blocs de contraintes. Nous désignons par SR l'ensemble de règles d'inférence excepté la Règle [5.4]. Nous calculons les trois systèmes de contraintes S_1, S_2 et S_3.

Phase 1 :	*S_1, la clôture de S par SR excepté les Règles [5.26] et [5.27] ;*
Phase d'étiquetage :	*S_2, la clôture de S_1 par les Règles [5.4] et [5.5]* uniquement *;*
Phase 2 :	*S_3, la clôture de S_2 par SR. Cette clôture est notée $(S)\downarrow$.*

La phase d'étiquetage ajoute des étiquettes pour créer les contraintes maîtres (Règle [5.4]), tout en s'assurant que nous favorisons toujours les contraintes d'égalité à celles de type $Forge_c$ (Règle [5.5]). Bien que les phases 1 et 2 soient similaires vis à vis des règles utilisées, leur attitude diffère. En effet, pour une étape $R_i \Rightarrow S_i$, la Phase 1 ne serait jamais utilisée pour un entrelacement de contraintes utilisant une contrainte maître $\mathcal{M}(\overrightarrow{X})$ où $\mathcal{L}(X) = E_{i-1}$. Ceci signifie que, pendant la Phase 1, les variables ayant un niveau maximum (le niveau de variables actuel), ne peuvent pas encore être remplacées d'un bloc à l'autre, pour la simple raison qu'aucune contrainte maître n'a été déclarée pour ces variables. Cependant, pour la Phase 2, nous n'avons pas cette limitation puisque celle-ci est précédée de la phase d'étiquetage et donc, même les variables du niveau courant ont des contraintes maîtres.

La normalisation du système de contrainte est utilisée dans notre procédure de vérification. Cette procédure commence par choisir une exécution possible du protocole qui est déduite du choix de l'ordre d'exécution π. Elle teste à la fin de l'exécution si le secret Sec est dérivable par l'intrus pour une certaine longueur e du $mpair(,)$. Pour ce faire, nous considérons l'exécution choisie au départ en ayant comme dernier terme à construire par l'intrus, le secret Sec. Le test

de dérivabilité du secret est alors équivalent au test de satisfaisabilité du système de contraintes normalisé déduit à la fin de cette exécution. Pour obtenir ce système de contraintes normalisé de l'exécution considérée, nous commençons par initialiser le système en le mettant à vide. La normalisation du système de contrainte est effectuée étape par étape. Chaque contrainte ajoutée à l'une de ces étapes est mise en conjonction avec le système de contraintes normalisé déduit de la normalisation des contraintes des étapes précédentes. Les contraintes sont ajoutées selon l'ordre π. Pour chacune de ces étapes, nous supposons que les contraintes maîtres et sous-maîtres ont déjà été calculées. L'environnement utilisé au cours de la normalisation est alors celui qui contient toutes les contraintes maîtres du système de contraintes. Puisque l'environnement dépend aussi des blocs, pour chaque bloc du système de contraintes, l'environnement contient également les contraintes sous-maîtres du bloc en question à l'exception des contraintes qui n'ajouterait pas d'informations utiles dans le cas de leurs utilisations, i.e. lors des remplacements des variables. Ces contraintes sont de la forme "variable égale à une autre variable". Notons que, que ce soit pour les contraintes maîtres ou sous-maîtres, ces contraintes concernent uniquement des variables de niveau strictement inférieur au niveau courant (celui de l'étape en cours de traitement). Toute cette procédure de vérification de protocoles bien tagués avec clefs autonomes est définie par l'Algorithme 6.2.

Algorithme 6.2 : Algorithme de vérification des protocoles bien tagués avec clefs autonomes. Soit $P = \{R'_\imath \Rightarrow S'_\imath | \imath \in J\}$ un protocole bien tagué avec clefs autonomes, $Sec \in \mathcal{T}$, et $S_0 \subset \mathcal{T}_g$.

1. Choisir un ordre d'exécution $\pi : J \rightarrow 1..k$.
2. Soit $R_i \triangleq R'_{\pi^{-1}(i)}$ et $S_i \triangleq S'_{\pi^{-1}(i)}$ $\forall i \in 1..k$. Soit $R_{k+1} \triangleq Sec$.
3. Soit $CBS_0 \triangleq \forall Q\, \exists R\, \top$, avec $Q = R = \emptyset$, le système de contraintes initial.
4. Pour i de 1 à $k+1$:
   (a) Supposons que $CBS_{i-1} \triangleq \forall Q \exists R\ (B_1, \mathcal{E}_{i-1,1}) \vee \ldots \vee (B_p, \mathcal{E}_{i-1,p})$;
   (b) Soit $ctr_i \triangleq R_i \in Forge(S_0, S_1, .., S_{i-1}, \emptyset)$;
   (c) Soit $\mathcal{E}_i = \bigcup_{\overrightarrow{X}} \mathcal{M}(CBS_{i-1}, \overrightarrow{X})$ et pour tout $j = 1, ..p$, $X, Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, $\mathcal{E}_{i,j} = \lceil \mathcal{E}_i \cup \mathcal{SM}(B_j, \mathcal{X}) \rceil \backslash \{(X = Y)\}$;
   (d) Soit $CBS_i \triangleq (\forall Q \exists R\ (B_1 \wedge ctr_i, \mathcal{E}_{i,1}) \vee .. \vee (B_p \wedge ctr_i, \mathcal{E}_{i,p}))\downarrow$
5. Tester la satisfaisabilité de CBS_{k+1} (répondre *protocole pas sur* ssi satisfaisabilité).

Dans l'Algorithme 6.2, les ensembles $\mathcal{E}_i, \mathcal{E}_{i,j}$ désignent respectivement l'ensemble des contraintes maîtres pour les vecteurs des variables et l'ensemble des contraintes sous-maîtres pour les variables du bloc B_j. Ces deux ensembles concernent des variables d'un niveau strictement inclus dans E_{i-1} (strictement inférieur au niveau de l'étape courante). La notation $\top$ représente la valeur *vrai*. CBS_0 désigne le système de contraintes initial. CBS_i désigne le système de contraintes correspondant à toutes les étapes de 0 à i. Finalement, ctr_i correspond à la contrainte relative à l'étape i de l'exécution choisie.

Nous énonçons maintenant les résultats trouvés pour cette procédure de vérification. Tout d'abord, quand elle est appliquée à des protocoles sans indices et sans *mpair* pouvant générer des indices, notre procédure termine. Ce résultat justifie le fait que notre modèle est bel et bien une extension des modèles classiques tel que [89]. Ce résultat est énoncé en Proposition 6.2.0.6.

Proposition 6.2.0.6 *Terminaison pour les protocoles sans indices et sans mpair(,).*
L'algorithme 6.2 termine pour les protocoles sans variables d'indices et sans mpair(,).

La preuve de la Proposition 6.2.0.6 sera détaillée en Section 6.5. Ensuite, nous avons au Lemme 6.2.0.7 la correction et complétude de la normalisation, i.e. de notre système d'inférence.

Lemme 6.2.0.7 *Correction et complétude de la normalisation.*
Soient CBS_i et ctr_i ($i = 1..k+1$) le système de contraintes et la contrainte de l'étape i définis dans l'algorithme 6.2, pour un certain protocole P bien tagué avec clefs autonomes. Alors, pour tout e, $[\![CBS_{i-1} \wedge ctr_i]\!]^e_\emptyset = \left|\!\left[(CBS_{i-1} \wedge ctr_i)\downarrow \right]\!\right|^e_\emptyset$

La preuve du Lemme 6.2.0.7 est détaillée en Section 6.4. En outre, notre normalisation termine même pour la classe des protocoles bien tagués avec clefs autonomes. Ce résultat fait l'objet de Lemme 6.2.0.8.

Lemme 6.2.0.8 *Terminaison de la normalisation.*
L'Algorithme 6.2 termine pour les protocoles bien tagués avec clefs autonomes.

Finalement, une fois qu'un système de contraintes est normalisé, on peut en tester la satisfaisabilité, i.e. il existe une solution ou pas, et ainsi donc savoir s'il existe une attaque ou si le protocole est sûr. Ceci est énoncé par le Lemme 6.2.0.9.

Lemme 6.2.0.9 *Satisfaisabilité de la forme normale.*
Quand l'Algorithme 6.2 est appliquée à un protocole P bien-tagué avec clefs autonomes, la satisfaisabilité du système de contraintes normalisé peut être décidée.

Le lemme 6.2.0.9 est prouvé en Section 6.7. Il est à noter que, pour la version du protocole Asokan-Ginzboorg sans clefs autonomes, sa vérification a terminé et le test de satisfaisabilité de CBS_{k+1} a été trivial. Par conséquent, Les Lemmes 6.2.0.7, 6.2.0.8 et 6.2.0.9 montrent bien que l'Algorithme 6.2 constitue bien une procédure de décision pour les protocoles bien tagués avec clefs autonomes. Ce principal résultat est énoncé en Théorème 6.2.0.10.

Théorème 6.2.0.10 *Décidabilité de la vérification de protocoles bien tagués avec clefs autonomes.*
Le problème d'insécurité pour les protocoles bien tagués avec clefs autonomes est décidable.

6.3 Quelques définitions pour les preuves

Avant d'entamer les preuves des résultats énoncés dans la Section 6.2, nous revenons sur la définition des dérivations de la Section 5.4.4 du Chapitre 5 pour définir les dérivations non redondantes comme suit :

Définition 6.3.0.11 *Dérivation non-redondante.*
Soit une dérivation $D = E_0 \longrightarrow_{L_1} \ldots \longrightarrow_{L_l} E_l$ ayant pour but u. D est une dérivation non redondante si $\forall i \forall t \in E_i$, si $L_c(t) \in D$ alors $\nexists L_d(\_) \in D$ générant t, et si $\exists L_d(\_) \in D$ générant t alors $L_c(t) \notin D$. Nous désignons par NRD l'ensemble de dérivations non redondantes.

Remarque 6.3.0.12 *Pour chaque dérivation $D_t(E)$, il existe une dérivation non redondante $D'_t(E)$. En effet, D' est obtenue par élimination de $L_c(t)$ si $L_d(\_) \in D$ où $L_d(\_)$ génères t et en éliminant chaque $L_d(\_) \in D$ qui génères t si $L_c(t) \in D$.*

Les définitions des prédicats Dy, Dy_c et Dy_d de la Définition 5.4.4.1 n'utilisent plus des dérivations quelconques mais utilisent plutôt des dérivations non redondantes. Dans toutes les preuves qui suivent, nous utilisons des dérivations non redondantes.

6.4 Correction et complétude

Le but de cette section est de montrer que nos différentes règles d'inférence conservent l'ensemble des solutions du système de contraintes initial. En effet, nous disons qu'une règle $F_1 \rightarrow F_2$, applicable à un système de contraintes, est complète quand $\forall e$, $[\![F_1]\!]^e_\emptyset \subseteq [\![F_2]\!]^e_\emptyset$. Cette même règle est dite correcte quand $\forall e$, $[\![F_1]\!]^e_\emptyset \supseteq [\![F_2]\!]^e_\emptyset$. Notons aussi pour une règle r, post(r) désigne la partie droite de r et pre(r) désigne la partie gauche de r.

Nous commençons tout d'abord, en Section 6.4.1, par déceler quelques propriétés qui sont conservées par nos règles de résolution de contraintes. Ces propriétés seront utilisées pour les preuves de la correction et complétude de nos règles d'inférence. Ces preuves sont présentées en Section 6.4.2.

6.4.1 Propriétés de notre système d'inférence

Nous prouvons dans cette section des invariants exprimant certaines propriétés satisfaites par des termes ou par des contraintes se produisant dans certains systèmes de contraintes dérivés à une certaine étape de la normalisation.

Nous commençons par le premier invariant (Invariant 6.4.1.1), selon lequel tout terme quelconque de notre système de contraintes contient au plus une seule variable d'indice. Cet invariant sera utilisé pour la preuve de correction et complétude de la Règle [5.35].

Invariant 6.4.1.1 $\#Var_{\mathcal{I}}(t) \leq 1$ *pour* $t \in \mathcal{T}$.

Preuve. Initialement, les contraintes sont de la forme $t \in Forge(E, \mathcal{K})$. Selon la propriété d'autonomie de *mpair* sur $\mathcal{P}$ (voir Définition 5.6.2.1), on a $Var_{\mathcal{I}}(t) = \emptyset$. Ainsi, l'Invariant 6.4.1.1 est valide pour les contraintes initiales. Nous montrons que l'application des règles de SR conserve cet invariant.
Les Règles [5.1], [5.4], [5.5], [5.6] et [5.8] ne changent aucune contraintes. L'Invariant 6.4.1.1 reste donc valide.
La Règle [5.2] change une contrainte en une autre tout en conservant les mêmes termes des contraintes initiales. Ainsi, l'Invariant 6.4.1.1 est satisfait. La Règle [5.3] élimine tout le bloc. L'invariant reste donc valide. La Règle [5.7] ajoute une nouvelle contrainte $Y_j = Z$ au bloc. Or, cette contrainte satisfait l'invariant.
La Règle [5.9] transforme une contrainte $t \in Forge(E, \mathcal{K})$ soit en une contrainte $t \in Forge_c(E, \mathcal{K})$ en préservant le même t, ou en une contrainte de type Sub en utilisant les deux termes t et $w \in E$. Or, par hypothèse d'induction, $\#Var_{\mathcal{I}}(t) \leq 1$. En outre, selon la propriété d'autonomie de *mpair* sur $\mathcal{P}$, on a $Var_{\mathcal{I}}(w) = \emptyset$, et donc $\#Var_{\mathcal{I}}(w) = 0$. Ainsi, l'invariant est satisfait.
La Règle [5.10] décompose le terme à composer $\langle t_1, \ldots, t_m \rangle$ en ses sous-termes t_i, $i = 1 \ldots m$.
Vu que $\#Var_{\mathcal{I}}(\langle t_1, \ldots, t_m \rangle) \leq 1$ (par hypothèse d'induction), on a $\#Var_{\mathcal{I}}(t_i) \leq 1$ $\forall i = 1 \ldots m$.
Le même raisonnement est appliqué pour les Règles [5.11] and [5.12].
Pour la Règle [5.13], grâce à la propriété d'autonomie de *mpair*, on a $Var_{\mathcal{I}}(t) = \{k\}$. L'invariant est donc valide.
La Règle [5.14] élimine tout le bloc. L'invariant reste donc valide. La Règle [5.15] transforme une contrainte de type Sub soit en une autre contrainte Sub, soit en une contrainte d'égalité tout

en conservant les deux termes utilisés dans le Sub (t et u). L'invariant est donc valide dans les deux cas.
Pour la Règle [5.16], la contrainte Sub est transformée en une contrainte de type Sub avec le même terme t et le sous-terme t_i de $\langle t_1, \ldots, t_m \rangle$. Vu que $\#Var_{\mathcal{I}}(\langle t_1, \ldots, t_m \rangle) \leq 1$ (par hypothèse d'induction), on a $\#Var_{\mathcal{I}}(t_i) \leq 1 \ \forall i = 1 \ldots m$.
Un raisonnement similaire est valide pour les Règles [5.17] et [5.18] (avec l'ajout d'une contrainte de type $Forge$ utilisant un sous-terme du terme initial t).
Pour la Règle [5.19], grâce à la propriété d'autonomie de $mpair$, on a $Var_{\mathcal{I}}(u) = \{k\}$. La Règle [5.20] élimine tout le bloc. Ainsi, l'invariant est valide pour ces deux règles. Les Règles [5.21] et [5.22] éliminent soit une contrainte ($\top$) soit un bloc ($\bot$). L'invariant reste donc satisfait.
La Règle [5.23] transforme une contrainte d'égalité de deux termes t et t' en des contraintes d'égalité des sous-termes de t et t'. L'invariant est donc valide.
Pour la Règle [5.24], selon la propriété d'autonomie de $mpair$, on a $Var_{\mathcal{I}}(u) = \{k\}$ et $Var_{\mathcal{I}}(v) = \{l\}$. Ainsi, l'invariant est valide.
La Règle [5.25] remplace une variable d'indice par une autre dans un bloc. L'invariant reste donc satisfait.
Pour les Règles [5.26], [5.27], [5.28], [5.29], [5.30] et [5.31], le système de contraintes résultant de l'application de ces règles a les mêmes termes que le système avant application de ces règles. De même, les Règles [5.33], [5.34] et [5.35], transforment un système de contraintes en un autre construit sur les mêmes termes que le système initial modulo un remplacement d'indices. Enfin, la Règle [5.32] élimine un bloc. L'invariant est alors satisfait par ces différents règles. □

La deuxième propriété exprime la conservation d'au moins une contrainte pour chaque variable, une fois que cette variable a déjà admis une contrainte. Cette propriété est représentée par l'Invariant 6.4.1.2 qui sera utilisé pour les preuves des corollaires 6.4.1.9 et 6.4.1.10.

Invariant 6.4.1.2 *Nous disons qu'une contrainte ctr est une contrainte pour X (ou X admet une contrainte ctr) si $ctr = (X \in Forge_c(E, \mathcal{K}))$ ou $ctr = (X = u)$.*
$\forall X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$, pour toute règle r dans $SR \cup \{Rule\ [5.4]\}$, si X admet une contrainte dans $pre(r)$, alors elle admet une contrainte dans tout bloc B de $post(r)$.

Preuve. Nous montrons que l'application des règles de $SR \cup \{Rule\ [5.4]\}$ conservent l'Invariant 6.4.1.2.
La Règle [5.2] transforme une contrainte pour Y en une autre contrainte pour Y sans éliminer les autres contraintes. Ainsi, nous avons toujours des contraintes pour X et Y. La Règle [5.3] élimine tout le bloc. L'invariant reste donc valide. Les Règles [5.4], [5.5], et [5.8] ne changent pas de contraintes. La Règle [5.6] transforme une contrainte pour X en une autre contrainte pour X sans éliminer d'autres contraintes. Ainsi, nous avons toujours des contraintes pour X et Y_j. La Règle [5.7] transforme une contrainte pour X en une autre contrainte pour X, tout en ajoutant une nouvelle contrainte pour Y_j. Nous concluons que les règles du premier groupe satisfont l'invariant.
La Règle [5.9] peut ajouter une contrainte pour une variable mais ne peut pas éliminer d'autres. Quant aux autres règles du groupe G_2, elles ne manipulent pas de contraintes pour des variables. L'invariant reste donc valide. La Règle [5.15] peut ajouter de contraintes pour une variable mais ne peut pas éliminer d'autres. Quant aux autres règles du groupe G_3, elles ne manipulent pas de contraintes pour des variables. L'invariant reste donc valide. La Règle [5.23] peut ajouter de contraintes pour une variable mais ne peut pas éliminer d'autres. Quant aux autres règles du groupe G_4, elles ne manipulent pas de contraintes pour des variables. L'invariant reste donc

valide.
Les Règles [5.26], [5.27], [5.30] et [5.31] n'éliminent pas de contraintes pour des variables. La Règle [5.28] élimine une contrainte pour X_i ($X_i \in Forge_c(E', \mathcal{K})$) mais il reste dans le même bloc une autre contrainte pour X_i : $X_i = u$. Nous raisonnons d'une manière similaire pour la Règle [5.29]. La Règle [5.32] élimine le bloc. Nous concluons que les règles du groupe G_5 satisfont l'Invariant 6.4.1.2.
Pour la Règle [5.34], la contrainte pour X_m : $X_m \in Forge_c(E', \mathcal{K})$ serait transformée soit en $X_m \in Forge_c(E', \mathcal{K})$ soit en $X_m = u_0\delta_0$. Dans les deux cas, il reste toujours une contrainte pour X_m. Pour la Règle [5.35], la contrainte pour X_m : $X_m = v$ est soit conservée soit transformée en une autre contrainte : $X_m = u_0\delta_0$. Dans les deux cas, il reste encore une contrainte pour X_m. Nous concluons que les règles du groupe G_6 satisfont l'Invariant 6.4.1.2. L'invariant est donc valide par application des règles de $SR \cup \{Rule\ [5.4]\}$. □

La troisième propriété exprime que le niveau des variables à l'intérieur du *Sub* est inclus dans l'ensemble de connaissances considéré dans le *Sub*. Cette propriété est illustrée par l'Invariant 6.4.1.3 qui sera utilisé pour la preuve de la Proposition 6.4.2.25.

Invariant 6.4.1.3 *Pour une contrainte* ($t \in Sub_\bullet(w, E, \mathcal{E}, \mathcal{K})$), *nous avons* $\forall X \leq w$ *où* $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, $\mathcal{L}(X) \subset E$.

Preuve. Initialement, les contraintes de type *Sub* sont obtenues à partir des contraintes de type *Forge* par la Règle [5.9]. Dans ce cas, nous obtenons la contrainte $t \in Sub(w, E, \mathcal{E}, \mathcal{K})$ où $w \in E$. Ainsi, $\forall X \leq w$, selon la notion d'exécution correcte, et par définition de $\mathcal{L}(X)$, nous avons $\mathcal{L}(X) \subset E$. Nous nous focalisons uniquement sur les règles manipulant des contraintes de type *Sub*. La Règle [5.15] transforme une contrainte $t \in Sub(u, E, \mathcal{E}, \mathcal{K})$ en la contrainte $t \in Sub_d(u, E, \mathcal{E}, \mathcal{K})$ sans modifier ni u ni E. L'Invariant 6.4.1.3 reste donc valide. Pour la Règle [5.16], vu que l'invariant est satisfait pour $\langle t_1, \ldots, t_m \rangle$ et E, alors il le reste pour un sous-terme de $\langle t_1, \ldots, t_m \rangle$: t_i et E. Le même raisonnement est valide pour les Règles [5.17], [5.18] et [5.19]. Les Règles [5.20] et [5.32] éliminent tout le bloc. L'invariant reste donc valide. La Règle [5.31] produit une contrainte $t \in Sub(w, E', \mathcal{E}, \mathcal{K})$ où w est donnée par la contrainte $X = w$ qui appartient à $\mathcal{E}$. Cependant, par construction de $\mathcal{E}$, $\mathcal{L}(w) \subset E'$. Enfin, le raisonnement est similaire pour la Règle [5.33]. □

Pour toute la suite, les propriétés et les lemmes introduits sont limités à la première phase du calcul d'un certain CBS_i. Celle-ci est donc effectuée avant la phase d'étiquetage des contraintes maîtres du niveau E_{i-1}.

La première propriété concernant cette première phase du calcul de CBS_i se focalise sur la comparaison des niveaux des variables existants dans les deux termes d'une certaine contrainte *Sub*. Elle exprime que le niveau d'une variable à l'intérieur du *Sub* est inclus dans le niveau d'une variable à l'extérieur du *Sub*. Cette propriété est énoncée par l'Invariant 6.4.1.4. Cet invariant sera utilisé pour la preuve de la Proposition 6.4.1.7.

Invariant 6.4.1.4 $\forall ctr$ *du calcul d'un certain* CBS_i *au cours de la phase 1,* $\forall X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ *t.q.* $\mathcal{L}(X) = E_{i-1}$, *si* $ctr = t \in Sub_\bullet(w, E, \mathcal{E}, \mathcal{K})$ *t.q.* $X \leq t$ *et* $\forall Y \leq w$ *alors* $\mathcal{L}(Y) \subset \mathcal{L}(X)$.

Preuve. Initialement, les contraintes sont de type *Forge*, ce qui valide l'invariant. Nous montrons que l'Invariant 6.4.1.4 est valide après chaque application des règles d'inférence. Le premier groupe ne manipule pas de contraintes de type *Sub*. L'invariant est donc valide. La Règle [5.9] génère une contrainte $t \in Sub(w, E, \mathcal{E}, \mathcal{K})$ où $w \in E$. Cependant, selon la notion d'exécution

correcte, et par définition de $\mathcal{L}(Y)$, $E \subset E_i$ et $\mathcal{L}(Y) \subseteq E$. Ainsi, $\mathcal{L}(Y) \subset \mathcal{L}(X)$, ce qui satisfait l'invariant. Les autres règles du groupe G_2 ne manipulent pas de contraintes de type Sub. Les règles du groupe G_2 satisfont donc l'invariant. La Règle [5.15] transforme une contrainte de type Sub en une autre de type Sub_d tout en conservant les mêmes termes t et u, ce qui satisfait l'invariant. La Règle [5.20] élimine tout le bloc. Les autres règles du groupe G_3 décomposent le terme à l'intérieur de la contrainte Sub. L'invariant reste donc valide. Les règles du groupe G_4 ne manipulent pas de contraintes de type Sub. L'invariant reste donc valide. La Règle [5.31] génère une contrainte $t \in Sub_d(w, E, \mathcal{E}, \mathcal{K})$. Cependant, w provient d'une contrainte sous-maître $(X = w) \in \mathcal{E}$. Par construction de l'environnement, $\mathcal{L}(Y) \subset \mathcal{L}(X)$, ce qui satisfait l'invariant. Les autres règles du groupe G_5 ne manipulent pas de contraintes de type Sub. Ainsi, les règles du groupe G_5 satisfont l'invariant. La Règle [5.33] génère une contrainte $t \in Sub_d(u\delta, E', \mathcal{E}, \mathcal{K})$. Cependant, $u\delta$ provient d'une contrainte maître $X_i = u \in \mathcal{E}$. En outre, par construction de $\mathcal{E}$, $\mathcal{L}(Y) \subset \mathcal{L}(X)$, ce qui satisfait l'invariant. Les autres règles du groupe G_6 ne manipulent pas de contraintes de type Sub. Ainsi, les règles du groupe G_6 satisfont l'invariant. Nous concluons que l'Invariant 6.4.1.4 est valide après application de nos règles d'inférence. □

Pour exprimer la deuxième propriété concernant toujours cette première phase du calcul de CBS_i, nous introduisons la Définition 6.4.1.5 qui désigne les différentes formes de contraintes que l'on puisse avoir pour une variable X dans notre système de contraintes.

Définition 6.4.1.5 *Une contrainte ctr' est dite de type* (1) *ou* (1′) *ou* (2) *ou* (2′) *ou encore* (3) *pour une variable X si*

$$\begin{array}{lllll} & ctr' = (t \in Forge(E, \mathcal{K})) & \text{où } X \leq t, & (1) \\ ou & ctr' = (t \in Forge_c(E, \mathcal{K})) & \text{où } X \leq t, & (1') \\ ou & ctr' = (t \in Sub(u, E, \mathcal{E}, \mathcal{K})) & \text{où } X \leq t & (2) \\ ou & ctr' = (t \in Sub_d(u, E, \mathcal{E}, \mathcal{K})) & \text{o'u } X \leq t & (2') \\ ou & ctr' = (u = v) & \text{où } X \leq u \text{ ou } X \leq v & (3) \end{array}$$

La propriété suivante assure que, si une variable de niveau courant admet une contrainte d'un des types de la Définition 6.4.1.5, alors nous conserverons, dans tous les blocs résultant de toute règle d'inférence utilisée sur celle-ci, une contrainte de l'un de ces mêmes types.

Invariant 6.4.1.6 *Pour toute règle r dans SR à l'exception des Règles [5.26] et [5.27] (i.e. Règles de la phase 1), $\forall X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ t.q. $\mathcal{L}(X) = E_{i-1}$, si pre(r) contient une contrainte de type* (1), (1′), (2), (2′) *ou* (3) *pour X alors $\forall B \in post(r)$, B contient une contrainte de type* (1), (1′), (2), (2′) *ou* (3) *pour X.*

Preuve. Les Règles [5.3], [5.14], [5.20],[5.22] et [5.32] éliminent tout le bloc, ce qui valide l'invariant. Les Règles [5.1], [5.2], [5.4], [5.5] et [5.8] ne changent pas de contraintes. L'invariant reste donc valide.
Pour les Règles [5.6] et [5.7], les contraintes pour d'autres variables que Y_j ne changent pas. Pour la variable Y_j, il existe une contrainte de type (3). Ainsi, l'invariant est valide.
Pour la Règle [5.9], nous obtenons soit une contrainte de type (1′), soit une contrainte de type (2) pour la variable X. Pour les Règles [5.10], [5.11], [5.12] et [5.13], le terme t est décomposé. Cette règle génère une contrainte de type (1) pour X. Nous concluons que les règles du deuxième groupe satisfont l'invariant.
La Règle [5.15] génère soit une contrainte de type (3), soit une contrainte de type (2′) pour X. Pour les Règles [5.16], [5.17], [5.18] et [5.19], le terme t est décomposé. Ils génèrent de contraintes

de type (2) pour X. Nous concluons que les règles du troisième groupe satisfont l'invariant.
Pour les Règles [5.21] et [5.25], pre(r) ne contiennent pas de contraintes de type (1), (1′),(2),(2′) ou (3) pour X. Pour les Règles [5.23] et [5.24], le terme t est décomposé. Ils génèrent de contraintes de type (3) pour X. Nous concluons que les règles du quatrième groupe satisfont l'invariant.
La Règle [5.28] transforme la contrainte $(X_i \in Forge_c(E, \mathcal{K}))$ en une autre : $(u \in Forge_c(E, \mathcal{K}))$. Néanmoins, il existe encore une contrainte de type (3) pour la variable X_i qui est $(X_i = u)^{\bullet}$. Le même raisonnement est valide pour la Règle [5.29]. Pour la règle [5.30], il existe une contrainte de type (1′) pour A. Nous concluons que les règles du cinquième groupe satisfont l'invariant.
Les Règles [5.33] et [5.34] conservent la même contrainte donnée par pre(r). Le Règle [5.35] utilise les mêmes termes pour les contraintes d'égalité. Elle transforme une contrainte de type (3) pour les variables de v ou X_m en d'autres contraintes de même type. Nous concluons que les règles du sixième groupe satisfont l'invariant. □

Nous présentons maintenant la propriété qui caractérise la première phase de la normalisation. Elle exprime l'existence d'une contrainte de type $Forge$ ou de type égalité pour toute variable de notre système d'inférence à la fin de la première phase du calcul d'un certain CBS_i. Cette propriété est énoncée par la Proposition 6.4.1.7.

Proposition 6.4.1.7 *À la fin de la phase 1 dans le calcul de CBS_i, pour tout $X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$, pour tout bloc B, il existe une contrainte $ctr \in B$ telle que $ctr = (X \in Forge_c(E, \mathcal{K}))$ ou $ctr = (X = u)$.*

Preuve. Nous introduisons tout d'abord la prétention suivante :

Prétention 6.4.1.8 *$\forall ctr$ dans le calcul d'un certain CBS_i à la phase 1, $\forall X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$ t.q. $\mathcal{L}(X) = E_{i-1}$, si $ctr = (Y = u)$ et $X \leq u$ alors $\mathcal{L}(Y) < \mathcal{L}(X)$.*

Preuve. Nous montrons par contradiction que si $ctr = (Y = u)$ et $X \leq u$ alors $\mathcal{L}(Y) < \mathcal{L}(X)$. Soit $ctr = (Y = u)$ tel que $\mathcal{L}(Y) = \mathcal{L}(X)$. Considérons une dérivation $d = d'_j.L_j.d_j$ telle que $(Y = u) \in \text{post}(L_j)$. Ainsi, $\exists l < j$ tel que $(u' = v') \in \text{post}(L_l)$ où $Y \leq v'$ et $u \leq u'$ et pre(L_l) $\neq (u" = v")$ où $u' < u"$ et $v' < v"$. La contrainte $(u' = v')$ est obtenue soit en transformant une contrainte de type Sub en une contrainte d'égalité par la Règle [5.15], soit par entrelacement (Règle [5.35]). Notons que les Règles [5.33] et [5.34] génèrent des contraintes d'égalité mais qui ne suivent pas la forme de contraintes définies dans la Prétention 6.4.1.8. En effet, ces contraintes générées ne peuvent pas contenir X vu que ces contraintes appartiennent à $\mathcal{E}$ et $\mathcal{L}(X) = E_{i-1}$.
1. Cas où pre(L_l) $= (u' \in Sub(v', E, \mathcal{E}, \mathcal{K}))$.
Vu que $X < u'$, $\mathcal{L}(X) = E_{i-1}$ et qu'on calcule CBS_i, selon l'Invariant 6.4.1.4, $\mathcal{L}(Y) \subset \mathcal{L}(X)$, ce qui contredit l'hypothèse : $\mathcal{L}(Y) = \mathcal{L}(X)$. Le même raisonnement est valide si pre(L_l) $= (v' \in Sub(u', E, \mathcal{E}, \mathcal{K}))$.
2. Cas où $L_l = R$ [5.35], alors pre(L_l) $= (X_m = u')$ et comme contrainte maître, nous avons $(X_i = v')$. Vu que nous sommes dans la première phase, i.e. avant étiquetage des labels maîtres et sous-maîtres pour les variables de niveau E_{i-1}, alors $\mathcal{L}(v') \subset E_{i-1}$. Vu que $\mathcal{L}(X) = E_{i-1}$ et $Y < v'$ alors $\mathcal{L}(Y) \subset \mathcal{L}(X)$, ce qui contredit l'hypothèse de $\mathcal{L}(Y) = \mathcal{L}(X)$. □

Notons que, à la fin de la première phase du calcul de CBS_i, nous avons uniquement des contraintes résolues (Hypothèse H_1). Nous montrons par contradiction qu'il existe une contrainte ctr pour X dans chaque bloc du système de contraintes S. Soit B un bloc de S tel que $\nexists ctr \in B$

pour X (Hypothèse H_2). Selon l'Invariant 6.4.1.6, $\exists ctr' \in B$ tel que ctr' est de type (1), (1′), (2), (2′) ou (3) pour X. Nous distinguons cinq cas. Dans le premier cas, $ctr' = (t \in Forge(E, \mathcal{K}))$ où $X < t$, sinon (quand $t \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$) $ctr = ctr'$, ce qui contredit l'hypothèse H_1. Dans ce cas, la Règle [5.9] peut être appliquée, ce qui contredit l'hypothèse H_1.
Dans le deuxième cas, $ctr' = (t \in Forge_c(E, \mathcal{K}))$ où $X < t$. Les Règles [5.10], [5.11], [5.12] et [5.13] peuvent être alors appliquées, ce qui contredit l'hypothèse H_1.
Dans le troisième cas, $ctr' = (t \in Sub(u, E, \mathcal{E}, \mathcal{K}))$ où $X \leq t$. La Règle [5.15] peut alors être appliquée, ce qui contredit l'hypothèse H_1.
Dans le quatrième cas, $ctr' = (t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$ où $X \leq t$. Nous distinguons deux cas. Dans le premier cas, $u \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$. Les Règles [5.16], [5.17], [5.18], [5.19] et [5.20] peuvent alors être appliquées, ce qui contredit l'hypothèse H_1. Dans le deuxième cas, $u = Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$. Or, selon l'Invariant 6.4.1.4, $\mathcal{L}(Y) \subset \mathcal{L}(X)$. Ainsi, $\exists ctr_3 \in \mathcal{E}$ tel que $ctr_3 = (Y \in Forge(E3, \mathcal{K}'))$ ou $ctr_3 = (Y = u3)$ (par construction de $\mathcal{E}$). Les Règles [5.31], [5.32] et [5.33] peuvent alors être appliquées, ce qui contredit l'hypothèse H_1.
Dans le cinquième cas, $ctr' = (u = v)$ où $X \leq v$ ou $X \leq u$. Nous distinguons deux cas. Dans le premier cas, $u \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$. Les Règles [5.22], [5.23] et [5.24] peuvent alors être appliquées, ce qui contredit l'hypothèse H_1. Dans le deuxième cas, $u \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$. Si $u = X$ alors $ctr' = ctr$, ce qui contredit l'hypothèse H_2. Si $u = Y$ et $X \leq v$ alors, selon la Prétention 6.4.1.8, $\mathcal{L}(Y) \subset \mathcal{L}(X)$. Ainsi, $\exists ctr" \in \mathcal{E}$ pour Y et donc les Règles [5.27] et [5.35] peuvent être appliquées, ce qui contredit l'hypothèse H_1. □

Nous étendons le résultat énoncé par la Proposition 6.4.1.7 pour exprimer en Corollaire 6.4.1.9 l'existence d'une contrainte maître unique pour chaque variable et ceci à la fin de toute phase de la normalisation.

Corollaire 6.4.1.9 *À la fin de toute phase, pour tout bloc B, $\forall X_i \in \mathcal{X}_\mathcal{I}$, il existe une unique contrainte maître $(ctr)^m \in B$ telle que $ctr = (X_i \in Forge_c(E, \mathcal{K}))$ ou $ctr = (X_i = u)$.*

Preuve. Pour la première phase, c'est une conséquence immédiate de la Proposition 6.4.1.7. L'unicité de la contrainte maître est garantie par la condition $B \cap \mathcal{M}(\overrightarrow{X}) = \emptyset$ de la Règle [5.4]. Au début de la deuxième phase, pour un bloc B, $\forall X_i \in \mathcal{X}_\mathcal{I}$, il existe une seule contrainte $(ctr)^m \in B$ telle que $ctr = (X_i \in Forge_c(E, \mathcal{K}))$ ou $ctr = (X_i = u)$ vu que l'Invariant 6.4.1.9 est préservé par la première phase.
Selon l'Invariant 6.4.1.2, notre système de règles conserve des contraintes pour X_i (des contraintes de type $Forge$ ou d'égalité pour X_i) qui sont des contraintes maîtres candidates pour X_i. Nos règles n'éliminent jamais des contraintes maîtres pour les variables. Elles peuvent seulement changer des contraintes maîtres pour les variables par introduction de nouvelles contraintes candidates (de type égalité). Ceci est assuré par la Règle [5.5]. Cette règle procède à l'étiquetage d'une nouvelle contrainte maître pour $\overrightarrow{X}$ $(X_j = u)$ tout en éliminant le label de l'ancienne contrainte maître. Ainsi, il ne reste qu'une seule contrainte maître pour $\overrightarrow{X}$. □

Finalement, nous étendons le résultat donné par le Corollaire 6.4.1.9 pour exprimer l'existance d'une contrainte de type $Forge$ ou égalité pour toute variable non indicée de notre système de contraintes, et ceci à la fin de chaque phase. Cette propriété est énoncée par le Corollaire 6.4.1.10.

Corollaire 6.4.1.10 *À la fin de toute phase de la normalisation, pour tout bloc B, $\forall X \in \mathcal{X}$, il existe une contrainte $ctr \in B$ telle que $ctr = (X \in Forge_c(E, \mathcal{K}))$ ou $ctr = (X = u)$.*

Preuve. La preuve découle directement de la Proposition 6.4.1.7 et de l'Invariant 6.4.1.2. □

6.4.2 Correction et complétude des règles

Nous prouvons la correction et la complétude de chaque règle de notre système d'inférence pour une substitution d'indice τ et une valeur e pour n. Nous considérons les groupes de règles dans l'ordre de leurs définitions dans la Section 5.8 du Chapitre 5.

La correction et la complétude des Règles [5.1], [5.6], [5.7], [5.21] et [5.22] est triviale. Les Règles [5.4], [5.5] et [5.8] sont correctes et complètes vu que la sémantique d'une contrainte étiquetée est la même que pour la même contrainte sans étiquetage, et vu que ces règles modifient uniquement les étiquettes des contraintes.

Nous regroupons en Table 6.1 les différentes propositions prouvant la correction et complétude des différentes règles de notre système d'inférence. Dans cette table, nous associons à chacune des règles, la ou les propositions pour les preuves de sa correction et de sa complétude. Nous mentionnons aussi en troisième position les autres preuves intermédiaires dont ces propositions ont besoin. Notons que la correction et la complétude des autres règles qui ne figurent pas dans cette table sont triviales.

Proposition 6.4.2.1 *La Règle [5.2] est correcte et complète.*

Preuve.

$$\sigma \in [\![(X = u)^{sm} \wedge (Y = X)]\!]^e_\tau \quad \text{ssi} \quad \overline{X}^e\tau\sigma = \overline{u}^e\tau\sigma \wedge \overline{Y}^e\tau\sigma = \overline{X}^e\tau\sigma \quad \text{ssi}$$
$$\overline{X}^e\tau\sigma = \overline{u}^e\tau\sigma \wedge \overline{Y}^e\tau\sigma = \overline{u}^e\tau\sigma \quad \text{ssi} \quad \sigma \in [\![(X = u)^{sm} \wedge (Y = u)]\!]^e_\tau$$

□

Proposition 6.4.2.2 *La Règle [5.3] est correcte et complète.*

Preuve. Soit B un bloc avec $(X = u) \in B$, $Y < u$, et $Y \sqsubset X$ dans B. Ainsi, $\forall\tau$, $\forall e$, $\forall\sigma \in [\![B]\!]^e_\tau$, nous avons $\overline{Y}^e\tau\sigma < \overline{X}^e\tau\sigma$ grâce au fait que $Y \sqsubset X$, et $\overline{Y}^e\tau\sigma < \overline{X}^e\tau\sigma$ grâce au fait que $Y < u$ et $(X = u) \in B$. Or, ceci est impossible, et donc $[\![B]\!]^e_\tau = [\![\bot]\!]^e_\tau$ □

La correction et la complétude de la Règle [5.9] découlent des Propositions 6.4.2.3 et 6.4.2.6 ci-dessous.

Proposition 6.4.2.3
$[\![B \wedge t \in Forge(E, \mathcal{K})]\!]^e_\tau \subseteq \big|[\![(B \wedge t \in Forge_c(E, \mathcal{K})) \vee \bigvee_{w \in E} (B \wedge t \in Sub(w, E, \mathcal{E}, \mathcal{K}))]\!]\big|^e_\tau$.

Preuve. Soit $\sigma \in [\![B \wedge t \in Forge(E, \mathcal{K})]\!]^e_\tau$. Alors, $\sigma \in [\![B]\!]^e_\tau \cap [\![t \in Forge(E, \mathcal{K})]\!]^e_\tau$. Cependant, $[\![t \in Forge(E, \mathcal{K})]\!]^e_\tau = [\![t \in Forge_c(E, \mathcal{K})]\!]^e_\tau \cup ([\![t \in Forge(E, \mathcal{K})]\!]^e_\tau \setminus [\![t \in Forge_c(E, \mathcal{K})]\!]^e_\tau)$. Ainsi, nous allons montrer qu'il existe $w \in E$ tel que $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$ si $\sigma \in ([\![t \in Forge(E, \mathcal{K})]\!]^e_\tau \setminus [\![t \in Forge_c(E, \mathcal{K})]\!]^e_\tau)$. Nous prouvons cette proposition en utilisant quelques lemmes. Commençons par le Lemme 6.4.2.4 qui découle de la Proposition 2 de [89].

Lemme 6.4.2.4 *Soient $t \in Dy(E, \mathcal{K})$ et $\gamma \in Dy_c(E, \mathcal{K})$ données par $D_\gamma(E) \in NRD$. Alors, il existe une dérivation $D'_t(E) \in NRD$ vérifiant $L_d(\gamma) \notin D'$.*

Preuve. La preuve est exactement la même que celle mentionnée dans [89] à la différence que $D_\gamma(E)$ n'est pas une dérivation minimale mais plutôt une dérivation non-redondante. La preuve reste valide vu qu'il n'y a pas de règles inutiles dans $D_\gamma(E)$ puisqu'il s'agit d'une dérivation non-redondante. En outre, $D'_t(E)$ telle qu'elle est construite dans [89] est une dérivation générale. Néanmoins, selon la Remarque 6.3.0.12, il existe une dérivation non-redondante équivalente à $D'_t(E)$. □

Dans notre contexte, le Lemme 6.4.2.4 exprime que pour tout terme t, un ensemble de termes E, un ensemble de termes $\mathcal{K}$ et une variable $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, et pour tout e, τ et σ, si $\overline{t}^e\tau\sigma \in Dy(E\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et $X\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ alors il existe une dérivation D' construisant $\overline{t}^e\tau\sigma$ à partir de $\overline{E}^e\tau\sigma$ où $X\tau\sigma$ n'est jamais décomposé. Ceci peut être facilement généralisé du singleton $\{X\}$ à tout ensemble de variables.

Lemme 6.4.2.5 *Si $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, alors il existe $w \in E$ t.q. $\overline{t}^e\tau\sigma \leqslant^L_{F\sigma} \overline{w}^e\tau\sigma$ avec $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$ et $\forall X$, si $X\tau\sigma \in L$ alors $X\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$.*

Preuve. Supposons que $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Selon le Lemme 6.4.2.4 itéré comme décrit ci-dessus, il existe une dérivation D de $\overline{E}^e\tau\sigma$ à $\overline{t}^e\tau\sigma$ tel que $\forall X$, si $L_d(X\tau\sigma) \in D$ alors $X\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. En outre, $\overline{t}^e\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ par définition de $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, et donc, D finit par une règle de décomposition. Maintenant, par itération sur la longueur de D en commençant par cette dernière décomposition, et en suivant la même idée que le Lemme 2 dans [89], il existe $w' \in E\tau\sigma$ tel que $\overline{t}^e\tau\sigma \leqslant^L_{F\sigma} w'$ avec $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$ et $\forall X$, si $X\tau\sigma \in L$ alors $X\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Il s'agit d'un terme de $E\tau\sigma$ qui est décomposé jusqu'à $\overline{t}^e\tau\sigma$ sans utiliser aucun terme de $\overline{\mathcal{K}}^e\tau\sigma$ comme clef de décryption. Enfin, il existe $w \in E$ t.q. $w\tau\sigma = w'$ nécessairement, et le lemme est prouvé. □

Nous pouvons donc finir la preuve de la Proposition 6.4.2.3. Vu que $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ selon le Lemme 6.4.2.5, il existe $w \in E$ tel que $\overline{t}^e\tau\sigma \leqslant^L_{F\sigma} \overline{w}^e\tau\sigma$ avec $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$ et $\forall X$, si $X\tau\sigma \in L$ alors $X\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Nous prouvons que si $\overline{t}^e\tau\sigma \leqslant^L_{F\sigma} \overline{w}^e\tau\sigma$ sans aucun $X\tau\sigma \in L$ tel que $X\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, alors $\sigma \in [\![\, t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ par récurrence sur (l, d) avec l la longueur de $\overline{t}^e\tau\sigma \leqslant^L_{F\sigma} \overline{w}^e\tau\sigma$ et $d = 1$ si $w \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, sinon 0.

Cas de base : Soient $l = 0$ et une dérivation arbitraire d. Alors, $\overline{t}^e\tau\sigma = \overline{w}^e\tau\sigma$, et donc $\sigma \in [\![\, t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ selon la sémantique de Sub.

Étape d'induction : Supposons que la formule ci-dessus est vrai pour toute instance strictement plus petite que (l, d), avec $l \geq 1$. Nous distinguons deux cas :

-Soit $w \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, et donc il existe un sous-terme direct w' de w, et il existe G, H, L_1, L_2 tels que $\overline{t}^e\tau\sigma \leqslant^{L_1}_G \overline{w'}^e\tau\sigma \leqslant^{L_2}_H \overline{w}^e\tau\sigma$, avec $G \cup H \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et la longueur de $\overline{t}^e\tau\sigma \leqslant^{L_1}_G \overline{w'}^e\tau\sigma$ strictement plus petite que l. Ainsi, $\sigma \in [\![\, t \in Sub(w', E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$, et donc $\sigma \in [\![\, t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ selon la sémantique de Sub.

-Soit $w = X$ pour un certain $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, et donc $X\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ vu que $l \geq 1$ et par conséquent $X\tau\sigma \in L$. En effet, selon les Corollaires 6.4.1.9 et 6.4.1.10 et par construction de $\mathcal{E}$, nous savons que soit $\exists v \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ t.q. $(X = v) \in \mathcal{E}$, et $X\tau\sigma = \overline{v}^e\delta\tau\sigma$ soit $X\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Cependant, le dernier cas est impossible vu que nous avons déjà $X\tau\sigma \notin Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Ainsi, nous avons $\overline{t}^e\tau\sigma \leqslant^L_{F\sigma} \overline{v}^e\tau\sigma$ vu que $\overline{w}^e\tau\sigma = X\tau\sigma = \overline{v}^e\tau\sigma$. Il suit par induction que $\sigma \in [\![\, t \in Sub(v, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ vu que $v \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ et $(l, 0) < (l, 1)$. Ainsi, $\sigma \in [\![\, t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$ selon le deuxième cas de la définition de la sémantique du Sub. □

Proposition 6.4.2.6
$\big|\big[\, (B \wedge t \in Forge_c(E, \mathcal{K})) \vee \bigvee_{w \in E} (B \wedge t \in Sub(w, E, \mathcal{E}, \mathcal{K})) \,\big]\big|^e_\tau \subseteq [\![\, B \wedge t \in Forge(E, \mathcal{K}) \,]\!]^e_\tau$.

Preuve. Afin de prouver la Proposition 6.4.2.6, nous allons démontrer la Proposition 6.4.2.7 et le Lemme 6.4.2.8.

Proposition 6.4.2.7 *Si $\sigma \in [\![\, t \in Sub(X, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_\tau$, alors, soit $\overline{t}^e\tau\sigma = X\tau\sigma$, soit $\exists v, \delta, k, \tau'$ tel que $k \notin \{t, X, \mathcal{E}\}$, $X\tau\sigma = \overline{v}^e\delta\tau'\sigma$, $v\delta \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ et $\sigma \in [\![\, t \in Sub(v\delta, E, \mathcal{E}, \mathcal{K}) \,]\!]^e_{\tau'}$.*

PREUVE. Vu que $\sigma \in [\![t \in Sub(X, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$, alors, $\exists u\ \exists F, L$ t.q. $u \leqslant^L_F \overline{X}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Cependant, vu que $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, on a $u = \overline{X}^e\tau$. En outre, nous avons deux cas pour la sémantique du *Sub*. Dans le premier cas, $u\sigma = \overline{t}^e\tau\sigma$. Ainsi, $\overline{X}^e\tau\sigma = \overline{t}^e\tau\sigma$ et par la suite la Proposition 6.4.2.7 suit. Dans le deuxième cas, $\exists v, \delta, k, \tau'$ tel que $k \notin \{t, X, \mathcal{E}\}$, $X\tau\sigma = \overline{v}^e\delta\tau'\sigma$ et $\sigma \in [\![t \in Sub(v\delta, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$. Néanmoins, par construction de $\mathcal{E}$, $v \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, ce qui conclut la preuve. □

Lemme 6.4.2.8 *Si $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$, alors, $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$.*

PREUVE. Nous procédons par récurrence sur $|\overline{w}^e\tau\sigma|$.

Cas de base : $|\overline{w}^e\tau\sigma| = 1$. Ainsi, soit $w \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$ soit $w \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$. Si $w \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$, alors, $\overline{w}^e\tau\sigma = \overline{t}^e\tau\sigma$. Ainsi, $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$. Si $w = X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, et vu que $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$, alors, selon la Proposition 6.4.2.7, nous avons (1) soit (2) avec :

(1) $\exists v, \delta, k, \tau'$ t.q. $k \notin \{t, X, \mathcal{E}\}, X\tau\sigma = \overline{v}^e\delta\tau'\sigma, v\delta \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ et $\sigma \in [\![t \in Sub(v\delta, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$
(2) $\overline{t}^e\tau\sigma = X\tau\sigma$

Si (1) alors $v\delta \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$ vu que $|\overline{v}^e\delta\tau'\sigma| = |\overline{w}^e\tau\sigma| = 1$. Cependant, $\overline{v}^e\delta\tau'\sigma = \overline{t}^e\tau'\sigma$ et $\overline{t}^e\tau'\sigma = \overline{t}^e\tau\sigma$ vu que $k, i \notin Var_\mathcal{I}(t)$ et $\mathcal{D}om(\tau') = \mathcal{D}om(\tau) \cup \{k, i\}$ (par définition de la sémantique du *Sub*). Par conséquent, $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$ vu que $\overline{v}^e\delta\tau\sigma = \overline{t}^e\tau\sigma = \overline{w}^e\tau\sigma$.
Si (2) alors vu que $w = X$, nous avons $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$.

Étape d'induction : Supposons que le Lemme 6.4.2.8 est vrai pour des termes de taille strictement plus petite que $|\overline{w}^e\tau\sigma|$.
Vu que $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$, nous avons $\exists u\ \exists F, L$ tel que $u \leqslant^L_F \overline{w}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et u,F valident un des deux cas de la définition de la sémantique du *Sub* :

Dans le premier cas, $u\sigma = \overline{t}^e\tau\sigma$ et $u\sigma \leqslant^{L\sigma}_{F\sigma} \overline{w}^e\tau\sigma$ vu que $u \leqslant^L_F \overline{w}^e\tau$ (par itération sur u suivant la définition de $\leqslant\_$). En outre, nous avons $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$ puisque $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$.

Dans le deuxième cas, $\exists v, \delta, k, \tau'$ t.q. $k \notin Var_\mathcal{I}(t, w, \mathcal{E})$, $\sigma \in [\![t \in Sub(v\delta, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$ et $u\sigma = \overline{v}^e\delta\tau'\sigma$. Nous pouvons supposer que $v\delta \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$. Sinon, selon la Proposition 6.4.2.7, il existe un autre choix pour v, δ, k, τ' tel que $v\delta \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ soit $\overline{t}^e\tau\sigma = v\delta\tau\sigma$. Le dernier cas mène à $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$ vu que $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et $\overline{t}^e\tau\sigma \leqslant^{L\sigma}_{F\sigma} \overline{w}^e\tau\sigma$. Ainsi, supposons que $v\delta \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, nous distinguons deux cas.

- Dans le premier cas, $u = \overline{w}^e\tau$ et donc $\overline{v}^e\delta\tau\sigma = \overline{w}^e\tau\sigma$. Si $v\delta \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$ alors, $|\overline{v}^e\delta\tau\sigma| = 1$ qui découle du cas initial de récurrence. Sinon, $\exists w'$ tel que $v\delta = f(w')$ avec $f \in \mathcal{G}$. Soit $F_0 = \{b\}$ si $v\delta = \{w'\}_b$ et $F_0 = \emptyset$ sinon. Notons que $F_0 \subseteq F$. En outre, vu que $\sigma \in [\![t \in Sub(v\delta, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$, alors, $\exists u', \exists F'$ tel que $u' \leqslant^{L'}_{F'} \overline{v\delta}^e\tau'$, $\overline{\mathcal{K}}^e\tau'\sigma \cap F'\sigma = \emptyset$, $F'\sigma \subseteq Dy(\overline{E}^e\tau'\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$ et les deux possibilités pour u'. Nous distinguons deux cas : $u' = \overline{v\delta}^e\tau'$ ou $u' <^{L'}_{F'} \overline{v\delta}^e\tau'$.
  Dans le premier cas, vu que $v\delta \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, $\overline{v\delta}^e\tau'\sigma = \overline{t}^e\tau'\sigma = \overline{t}^e\tau\sigma$. Ainsi, $\overline{w}^e\tau\sigma = \overline{t}^e\tau\sigma$ et donc il suit que $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$.
  Dans le deuxième cas, $u' \leqslant^{L'}_{F'} \overline{w'}^e\tau' <^{L'}_{\overline{F_0}^e\tau'} \overline{v\delta}^e\tau'$. Ainsi, $\sigma \in [\![t \in Sub(w', E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$.
  Puisque $\left|\overline{w'}^e\tau'\sigma\right| < |\overline{w}^e\tau\sigma|$, il suit que $\overline{t}^e\tau'\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w'}^e\tau'\sigma\}, \overline{\mathcal{K}}^e\tau'\sigma)$.
  Notons que $\overline{E}^e\tau' = \overline{E}^e\tau = \overline{E}^e$.
  En outre, $\overline{w'}^e\tau'\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$ vu que $\overline{w'}^e\tau'\sigma <^{L'\sigma}_{\overline{F_0}^e\tau'\sigma} \overline{v\delta}^e\tau'\sigma$, $\overline{v\delta}^e\tau'\sigma = \overline{w}^e\tau\sigma$, $\overline{\mathcal{K}}^e\tau'\sigma \cap \overline{F_0}^e\tau'\sigma = \emptyset$ et $\overline{F_0}^e\tau'\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ ($\overline{F_0}^e\tau' \subseteq F'$).
  Ainsi, $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$.

– Dans le deuxième cas, $\overline{v}^e\delta\tau'\sigma <^{L\sigma}_{F\sigma} \overline{w}^e\tau\sigma$ et $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{v}^e\delta\tau'\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$ vu que $|\overline{v}^e\delta\tau'\sigma| < |\overline{w}^e\tau\sigma|$. En outre, $\overline{v}^e\delta\tau'\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$ selon la définition $\leqslant_{\_}$ et puisque $\overline{v}^e\delta\tau'\sigma <^{L\sigma}_{F\sigma} \overline{w}^e\tau\sigma$, $\overline{\mathcal{K}}^e\tau'\sigma \cap F\sigma = \emptyset$ et $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Ainsi, nous avons $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$.

□

Revenons à la preuve de la Proposition 6.4.2.6.
Soit $\sigma \in \big|\big[\, (B \wedge (t \in Forge_c(E,\mathcal{K})) \vee \bigvee_{w\in E}(B \wedge t \in Sub(w,E,\mathcal{E},\mathcal{K}))\,\big]\big|^e_\tau$.
Si $\sigma \in [\![\, B \wedge t \in Forge_c(E,\mathcal{K})\,]\!]^e_\tau$ alors $\sigma \in [\![\, B \wedge t \in Forge(E,\mathcal{K})\,]\!]^e_\tau$ par définition de $Forge$ et de $Forge_c$. Soit $\sigma \in \big|\big[\, \bigvee_{w\in E} B \wedge t \in Sub(w,E,\mathcal{E},\mathcal{K})\,\big]\big|^e_\tau$. Par conséquent, il existe $w \in E$ tel que $\sigma \in [\![\, t \in Sub(w,E,\mathcal{E},\mathcal{K})\,]\!]^e_\tau \cap [\![\, B\,]\!]^e_\tau$. Selon le Lemme 6.4.2.8, $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma \cup \{\overline{w}^e\tau\sigma\}, \overline{\mathcal{K}}^e\tau\sigma)$. Cependant, $\overline{w}^e\tau\sigma \in \overline{E}^e\tau\sigma$. Ainsi, $\overline{t}^e\tau\sigma \in Dy_d(\overline{E}^e\tau\sigma, \overline{t}^e\tau\sigma)$. Par conséquent, la Proposition 6.4.2.6 suit. □

Proposition 6.4.2.9 *Les Règles [5.10], [5.11] et [5.12] sont correctes et complètes.*

Preuve. La correction de la Règle [5.10] est triviale. Pour la complétude de la Règle [5.10], soit $\sigma \in [\![\, \langle t_1,\ldots,t_m\rangle \in Forge_c(E,\mathcal{K})\,]\!]^e_\tau$. Ainsi, $\langle \overline{t_1}^e\tau\sigma,\ldots,\overline{t_m}^e\tau\sigma\rangle \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ vu que nous avons $\overline{\langle t_1,\ldots,t_m\rangle}^e\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Par définition de Dy_c, $\exists D$ une dérivation ayant pour but $\langle \overline{t_1}^e\tau\sigma,\ldots,\overline{t_m}^e\tau\sigma\rangle$ sans utiliser aucun terme de $\overline{\mathcal{K}}^e\tau\sigma$ comme clef de décryption et terminant par une règle de composition. Ainsi, les sous-termes de $\langle \overline{t_1}^e\tau\sigma,\ldots,\overline{t_m}^e\tau\sigma\rangle$ doivent être dans D, et par conséquent, $\bigwedge_{i\leq m}(\overline{t_i}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma))$. Enfin, les preuves de correction et de complétude des Règles [5.11] et [5.12] sont similaires à celles de la Règle [5.10]. □

Afin de montrer la Proposition 6.4.2.11, nous prouvons tout d'abord le Lemme 6.4.2.10.

Lemme 6.4.2.10 $[\![\, mpair(k,t) \in Forge_c(E,\mathcal{K})\,]\!]^e_\tau = [\![\, \forall k\ t \in Forge(E,\mathcal{K})\,]\!]^e_\tau$.

Preuve. Soit $\sigma \in [\![\, mpair(k,t) \in Forge_c(E,\mathcal{K})\,]\!]^e_\tau$. Ainsi, $\overline{mpair(k,t)}^e\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ menant à $\langle \overline{\tau_{k,1}(t)}^e\tau\sigma,\ldots,\overline{\tau_{k,e}(t)}^e\tau\sigma\rangle \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Ainsi, $\sigma \in [\![\, \forall k\ t \in Forge(E,\mathcal{K})\,]\!]^e_\tau$ vu que $\bigwedge_{i\leq e}(\overline{\tau_{k,i}(t)}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma))$. □

Proposition 6.4.2.11 *La Règle [5.13] est correcte and complète.*

Preuve. Selon nos sémantiques, nous avons : $[\![\, \forall Q\ \exists R\ S \vee (B \wedge mpair(k,t) \in Forge(E,\mathcal{K}))\,]\!]^e_{\_}$
$= \bigcap_Q \bigcup_R [\![\, S \vee (B \wedge mpair(k,t) \in Forge(E,\mathcal{K}))\,]\!]^e_\tau$
$= \bigcap_Q \bigcup_R [\![\, S\,]\!]^e_\tau \cup ([\![\, B\,]\!]^e_\tau \cap [\![\, mpair(k,t) \in Forge(E,\mathcal{K})\,]\!]^e_\tau)$
$= \bigcap_Q \bigcup_R [\![\, S\,]\!]^e_\tau \cup ([\![\, B\,]\!]^e_\tau \cap [\![\, \forall kt \in Forge(E,\mathcal{K})\,]\!]^e_\tau)$ (selon le Lemme 6.4.2.10)
$= \bigcap_Q \bigcup_R \bigcap_k [\![\, S\,]\!]^e_{\tau'} \cup ([\![\, B\,]\!]^e_{\tau'} \cap [\![\, t \in Forge(E,\mathcal{K})\,]\!]^e_{\tau'})$ (vu que k est frais, $\tau' = \tau.[k \leftarrow n_k]$)
$= \bigcap_Q \bigcap_k \bigcup_R [\![\, S\,]\!]^e_{\tau'} \cup ([\![\, B\,]\!]^e_{\tau'} \cap [\![\, t \in Forge(E,\mathcal{K})\,]\!]^e_\tau)$ (selon l'autonomie des $mpair$)
$= \bigcap_{Q.k} \bigcup_R [\![\, S \vee (B \wedge t \in Forge(E,\mathcal{K}))\,]\!]^e_\tau$
$= [\![\, \forall Q.k\ \exists R\ S \vee (B \wedge t \in Forge(E,\mathcal{K}))\,]\!]^e_\tau$ □

La correction et la complétude de la Règle [5.14] sont triviales. La complétude et la correction de la Règle [5.15] découle de la Proposition 6.4.2.12 et la Proposition 6.4.2.13.

Proposition 6.4.2.12 *La Règle [5.15] est complète.*

PREUVE. Soit $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$ et $W = [\![(t = w) \vee (t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$. Nous prouvons que $\sigma \in [\![(t = w)]\!]^e_\tau$ si $w = \{v\}_b$ et $b \in \mathcal{K}$ soit $\sigma \in W$. Nous savons que $\exists u,\ \exists F, L$ tels que $u \leqslant^L_F \overline{w}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et les deux cas de la sémantique du Sub. Nous distinguons deux cas dépendant si ($w = \{v\}_b$ et $b \in \mathcal{K}$) ou pas :

- ($w = \{v\}_b$ et $b \in \mathcal{K}$). Par définition de $<_{\_}$, nous avons $\overline{w}^e\tau \in L$ et $\overline{b}^e \in F$. Cependant, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$. Ainsi, $u \leqslant^L_F \overline{w}^e\tau$ mène à $u = \overline{w}^e\tau$. Nous pouvons supposer que $w \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, sinon $u \leqslant^L_F w$ mène à $u <^L_F w$. Par conséquent, nous avons seulement le premier cas de la sémantique du Sub, qui est $u\sigma = \overline{t}^e\tau\sigma$, ce qui mène à $\overline{w}^e\tau\sigma = \overline{t}^e\tau\sigma$.
- ($w \neq \{v\}_b$ ou $b \notin \mathcal{K}$). Nous avons donc les deux cas de la sémantique du Sub :
  - Dans le premier cas, $u\sigma = \overline{t}^e\tau\sigma$. En outre, $u \leqslant^L_F \overline{w}^e\tau$. Ainsi, $u <^L_F \overline{w}^e\tau$, ce qui mène à $\sigma \in [\![(t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$ soit $u = \overline{w}^e\tau$. Pour le dernier cas, nous avons $u\sigma = \overline{w}^e\tau\sigma$. Néanmoins, $u\sigma = \overline{t}^e\tau\sigma$. Ainsi, $\overline{w}^e\tau\sigma = \overline{t}^e\tau\sigma$, ce qui mène à $\sigma \in [\![(t = w)]\!]^e_\tau$. Nous concluons que, pour les deux cas, $\sigma \in W$.
  - Le deuxième cas est le même pour les sémantiques de Sub ou Sub_d. Ainsi, $\sigma \in W$ vu que $\sigma \in [\![t \in Sub_d(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$.

La Règle [5.15] est donc complète. □

Proposition 6.4.2.13 *La Règle [5.15] est correcte.*

PREUVE. Nous distinguons deux cas selon si ($w = \{v\}_b$ et $b \in \mathcal{K}$) ou pas :

- Supposons que ($w = \{v\}_b$ et $b \in \mathcal{K}$). Soit $\sigma \in [\![(t = w)]\!]^e_\tau$. Ainsi, $\overline{w}^e\tau\sigma = \overline{t}^e\tau\sigma$. Soit $u = \overline{w}^e\tau$. Nous avons $u \leqslant^\emptyset_\emptyset \overline{w}^e\tau$, $\emptyset \subseteq Dy(\overline{E}^e\tau\sigma, \emptyset)$ et $u\sigma = \overline{w}^e\tau\sigma = \overline{t}^e\tau\sigma$. Ainsi, $\sigma \in [\![(t \in Sub(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$.
- Supposons que ($w \neq \{v\}_b$ ou $b \notin \mathcal{K}$). Soit $\sigma \in [\![(t = w) \vee (t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$. Il y a deux cas :
  - $\sigma \in [\![(t = w)]\!]^e_\tau$. Nous prouvons d'une manière similaire au premier cas (i.e. $w = \{v\}_b$ et $b \in \mathcal{K}$) que $\sigma \in [\![(t \in Sub(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$.
  - $\sigma \in [\![(t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$. Par définition de Sub_d (la même sémantique que Sub avec la différence que $u <^L_F \overline{w}^e\tau$ si $u\sigma = \overline{t}^e\tau\sigma$) et vu que, si $u <^L_F \overline{w}^e\tau$ alors nous avons $u \leqslant^L_F \overline{w}^e\tau$, nous avons donc $\sigma \in [\![(t \in Sub(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$.

La Règle [5.15] est donc correcte. □

Proposition 6.4.2.14 *La Règle [5.16] est correcte et complète.*

PREUVE. Soit $W = [\![\bigvee_{i=1\ldots m} (t \in Sub(t_i, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$.

Soit $\sigma \in [\![t \in Sub_d(\langle t_1, .., t_m\rangle, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. Ainsi, $\exists u,\ \exists F, L$ tels que $u <^L_F \overline{\langle t_1, .., t_m\rangle}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et les deux possibilités pour u (les deux cas de la sémantique du Sub). Ainsi, $\exists i = 1 \ldots m$ tel que $u \leqslant^L_F \overline{t_i}^e\tau <^{\overline{\langle t_1,..,t_m\rangle}^e\tau}_\emptyset \overline{\langle t_1, .., t_m\rangle}^e\tau$ vu que $u <^L_F \overline{\langle t_1, .., t_m\rangle}^e\tau$. Ainsi, $\sigma \in W$.

Soit $\sigma \in W$. Soit $i = 1 \ldots m$ tel que $\sigma \in [\![(t \in Sub(t_i, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$. Ainsi, $\exists u,\ \exists F, L$ tels que $u \leqslant^L_F \overline{t_i}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et les deux possibilités pour u (les deux cas de la sémantique du Sub). En outre, vu que $t_i <_\emptyset \langle t_1, .., t_m\rangle$, alors $\overline{t_i}^e\tau <^{\overline{\langle t_1,..,t_m\rangle}^e\tau}_\emptyset \overline{\langle t_1, .., t_m\rangle}^e\tau$. Ainsi, $u <^{L \cup \{\overline{\langle t_1,..,t_m\rangle}^e\tau\}}_F \overline{\langle t_1, .., t_m\rangle}^e\tau$ et $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, d'où $\sigma \in [\![t \in Sub_d(\langle t_1, .., t_m\rangle, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. □

Proposition 6.4.2.15 *Les Règles [5.17] et [5.18] sont correctes et complètes.*

PREUVE. Nous nous focalisons tout d'abord sur la Règle [5.17].
Soit $W = [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K}) \wedge b \in Forge(E, \mathcal{K} \cup \{b\})]\!]^e_\tau$.

Soit $\sigma \in [\![t \in Sub_d(\{w\}^s_b, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. Ainsi, $\exists u,\ \exists F, L$ t.q. $u <^L_F \overline{\{w\}^s_b}^e \tau$, $\overline{\mathcal{K}}^e \tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e \tau\sigma, \overline{\mathcal{K}}^e \tau\sigma)$ et les deux possibilités pour u (les deux cas de la sémantique du Sub). Vu que $u <^L_F \overline{\{w\}^s_b}^e \tau$, alors $u \leqslant^L_F \overline{w}^e\tau <^{\overline{\{w\}^s_b}^e \tau}_{\overline{b}^e\tau} \overline{\{w\}^s_b}^e \tau$. Ainsi, $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. En outre, vu que $u <^L_F \overline{\{w\}^s_b}^e \tau$, alors $\overline{b}^e\tau \in F$, et donc, $\overline{b}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Par minimalité de la dérivation ayant pour but $\overline{b}^e\tau\sigma$, celui-ci ne doit pas être utilisé comme clef de décryption et donc $\overline{b}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K} \cup \{b\}}^e\tau\sigma)$. Ainsi, $\sigma \in [\![b \in Forge(E, \mathcal{K} \cup \{b\})]\!]^e_\tau$. Par conséquent, $\sigma \in W$.

Soit $\sigma \in W$. Soit $\sigma \in [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. Ainsi, $\exists u,\ \exists F, L$ tels que $u \leqslant^L_F \overline{w}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et les deux possibilités pour u (les deux cas de la sémantique du Sub). En outre, vu que $\overline{w}^e\tau <^{\{\overline{\{w\}^s_b}^e\tau\}}_{\overline{b}^e\tau} \overline{\{w\}^s_b}^e\tau$, nous avons $u <^{L \cup \{\overline{\{w\}^s_b}^e\tau\}}_{F \cup \{\overline{b}^e\tau\}} \overline{\{w\}^s_b}^e\tau$. Cependant, $b \notin \mathcal{K}$ (selon la condition de la Règle [5.15]). Ainsi, $F\sigma \cap (\{\overline{b}^e\tau\sigma\} \cup \overline{\mathcal{K}}^e\tau\sigma) = \emptyset$. En outre, vu que $\sigma \in [\![b \in Forge(E, \mathcal{K} \cup \{b\})]\!]^e_\tau$, nous avons $\overline{b}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K} \cup b}^e\tau\sigma)$. Ainsi, $\overline{b}^e\tau\sigma \in Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, et par conséquent, $F\sigma \cup \{\overline{b}^e\tau\sigma\} \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Ainsi, $\sigma \in [\![t \in Sub_d(\{w\}^s_b, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$.

Enfin, la preuve de la correction et de la complétude de la Règle [5.18] est similaire à celle de la Règle [5.17]. □

Proposition 6.4.2.16 *La Règle [5.19] est correcte et complète.*

Preuve. Nous nous focalisons tout d'abord sur la complétude.
Soit $\sigma \in [\![t \in Sub_d(mpair(k, w), E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. Ainsi, $\exists u\ \exists F, L$ tel que $u <^L_F \overline{mpair(k, w)}^e \tau$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et les deux possibilités pour u. Par conséquent, $u <^L_F \langle \overline{\tau_{k,1}(w)}^e \tau\sigma, \ldots, \overline{\tau_{k,e}(w)}^e \tau\sigma \rangle$. Ainsi, $\exists x \in \{1, \ldots, e\}$ t.q. $u \leqslant^L_F \overline{\tau_{k,1}(w)}^e \tau\sigma$, ce qui mène à $\sigma \in \bigcup_{x=1\ldots e} [\![t \in Sub(w, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau,[k \leftarrow x]}$. Ainsi, $\sigma \in [\![\exists k\ (t \in Sub(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$.

Nous nous focalisons maintenant sur la correction de la Règle [5.19].
Soit $\sigma \in [\![\exists k\ (t \in Sub(w, E, \mathcal{E}, \mathcal{K}))]\!]^e_\tau$.
Ainsi, $\sigma \in \bigcup_{x=1\ldots e} [\![t \in Sub_d(w, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau,[k \leftarrow x]}$. Soit $x \in \{1, \ldots, e\}$ et $\tau' = \tau, [k \leftarrow x]$. Nous avons $\sigma \in [\![t \in Sub_d(w, E, \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$. Ainsi, $\exists u\ \exists F, L$ tels que $u \leqslant^L_F \overline{w}^e\tau'$, $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau'\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$ et les deux possibilités pour u. Cependant, $\overline{w}^e\tau' = \overline{\tau_{k,x}(w)}^e\tau$. Nous avons donc $u \leqslant^L_F \overline{\tau_{k,x}(w)}^e \tau <^{\langle \overline{\tau_{k,1}(w)}^e \tau, \ldots, \overline{\tau_{k,e}(w)}^e \tau \rangle}_{\emptyset} \langle \overline{\tau_{k,1}(w)}^e \tau, \ldots, \overline{\tau_{k,e}(w)}^e \tau \rangle$.
Ainsi, $u <^{L \cup \{\langle \overline{\tau_{k,1}(w)}^e \tau\sigma, \ldots, \overline{\tau_{k,e}(w)}^e \tau \rangle\}}_{F} \overline{mpair(k, w)}^e \tau$ et $\overline{\mathcal{K}}^e\tau\sigma \cap F\sigma = \emptyset$, ce qui conduit à $\sigma \in [\![t \in Sub_d(mpair(k, w), E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$. □

Proposition 6.4.2.17 *La Règle [5.20] est correcte et complète.*

Preuve. La correction de la Règle [5.20] est triviale. Soit $\sigma \in [\![t \in Sub_d(c, E, \mathcal{E}, \mathcal{K})]\!]^e_\tau$ où $c \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$. Ainsi, $\exists u\ \exists F, L$ tels que $u <^L_F \overline{c}^e\tau$, $\overline{\mathcal{K}}^e\tau\sigma \cup F\sigma = \emptyset$, $F\sigma \subseteq Dy(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ et les deux possibilités pour u. Dans le premier cas, $u\sigma = \overline{t}^e\tau\sigma$ et $u <^L_F \overline{c}^e\tau$. Cependant, la dernière condition est impossible vu que $c\tau \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$ ($\overline{c}^e\tau = c\tau$), et par conséquent $u = c\tau$. D'une manière similaire, le deuxième cas du Sub traite le cas où $u \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, ce qui contredit $u = c\tau \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$. Nous concluons que dans les deux cas, nous avons $\sigma \in [\![\bot]\!]^e_\tau$. □

Proposition 6.4.2.18 *La Règle [5.23] est correcte et complète.*

Preuve. Ceci découle directement de nos sémantiques : $\sigma \in [\![f(u_1, .., u_m) = f(w_1, .., w_m)]\!]^e_\tau$ ssi $\overline{f(u_1, .., u_m)}^e \tau\sigma = \overline{f(w_1, .., w_m)}^e \tau\sigma$ ssi $f(\overline{u_1}^e\tau\sigma, .., \overline{u_m}^e\tau\sigma) = f(\overline{w_1}^e\tau\sigma, .., \overline{w_m}^e\tau\sigma)$ ssi $\bigwedge_{i=1\ldots m}(\overline{u_i}^e\tau\sigma = \overline{w_i}^e\tau\sigma)$ ssi $\sigma \in [\![\bigwedge_{i=1\ldots m} u_i = w_i]\!]^e_\tau$. □

Afin de montrer la Proposition 6.4.2.20, nous prouvons d'abord le Lemme 6.4.2.19.

Lemme 6.4.2.19 $[\![\, mpair(k,u) = mpair(l,w) \,]\!]^e_\tau = [\![\, \forall k\, u = w\delta_{l,k} \,]\!]^e_\tau$.

PREUVE.

$$\begin{array}{ll}
\sigma \in [\![\, mpair(k,u) = mpair(l,w) \,]\!]^e_\tau & \text{ssi} \\
\overline{mpair(k,u)}^e \tau\sigma = \overline{mpair(l,w)}^e \tau\sigma & \text{ssi} \\
\langle \overline{\tau_{k,1}(u)}^e, \ldots, \overline{\tau_{k,v}(u)}^e \rangle \tau\sigma = \langle \overline{\tau_{l,1}(w)}^e, \ldots, \overline{\tau_{l,v}(w)}^e \rangle \tau\sigma & \text{ssi} \\
\langle \overline{\tau_{k,1}(u)}^e \tau\sigma, \ldots, \overline{\tau_{k,v}(u)}^e \tau\sigma \rangle = \langle \overline{\tau_{l,1}(w)}^e \tau\sigma, \ldots, \overline{\tau_{l,v}(w)}^e \tau\sigma \rangle & \text{ssi} \\
\bigwedge_{i=1,\ldots,e} (\overline{\tau_{k,i}(u)}^e \tau\sigma = \overline{\tau_{l,i}(w)}^e \tau\sigma) & \text{ssi} \\
\bigwedge_{i=1,\ldots,e} (\overline{\tau_{k,i}(u)}^e \tau\sigma = \overline{\tau_{k,i}(w\delta_{l,k})}^e \tau\sigma) & \text{ssi} \\
\bigwedge_{i=1,\ldots,e} (\overline{u}^e \tau\tau_{k,i}\sigma = \overline{w\delta_{l,k}}^e \tau\tau_{k,i}\sigma) & \text{ssi} \\
\sigma \in \bigcap_{i=1,\ldots,e} [\![\, u = w\delta_{l,k} \,]\!]^e_{\tau,\tau_{k,i}} & \text{ssi} \\
\sigma \in [\![\, \forall k\, u = w\delta_{l,k} \,]\!]^e_\tau &
\end{array}$$

□

Proposition 6.4.2.20 *La Règle [5.24] est correcte et complète.*

PREUVE. Selon nos définitions, nous avons : $[\![\, \forall Q\ \exists R\ S \vee (B \wedge (mpair(k,u) = mpair(l,w))) \,]\!]^e$
$= \bigcap_Q \bigcup_R [\![\, S \vee (B \wedge (mpair(k,u) = mpair(l,w))) \,]\!]^e_\tau$ (où τ assigne des valeurs à Q et R)
$= \bigcap_Q \bigcup_R ([\![\, S \,]\!]^e_\tau \cup ([\![\, B \,]\!]^e_\tau \cap [\![\, mpair(k,u) = mpair(l,w) \,]\!]^e_\tau))$
$= \bigcap_Q \bigcup_R ([\![\, S \,]\!]^e_\tau \cup ([\![\, B \,]\!]^e_\tau \cap [\![\, \forall k\, u = w\delta_{l,k} \,]\!]^e_\tau))$ (selon le Lemme 6.4.2.19)
$= \bigcap_Q \bigcup_R \bigcap_k ([\![\, S \,]\!]^e_{\tau'} \cup ([\![\, B \,]\!]^e_{\tau'} \cap [\![\, u = w\delta_{l,k} \,]\!]^e_{\tau'}))$ (vu que k est frais, $\tau' = \tau.[k \leftarrow n_k]$)
$= \bigcap_Q \bigcap_k \bigcup_R ([\![\, S \,]\!]^e_{\tau'} \cup ([\![\, B \,]\!]^e_{\tau'} \cap [\![\, u = w\delta_{l,k} \,]\!]^e_{\tau'}))$ (selon l'autonomie des $mpair$, $Var_{\mathcal{I}}(u) \cap R = Var_{\mathcal{I}}(w) \cap R = \emptyset$))
$= \bigcap_{Q.k} \bigcup_R ([\![\, S \vee (B \wedge (u = w\delta_{l,k})) \,]\!]^e_{\tau'}) = [\![\, \forall Q.k\ \exists R\ S \vee (B \wedge (u = w\delta_{l,k})) \,]\!]^e_\tau$. □

Proposition 6.4.2.21 *La Règle [5.25] est correcte et complète.*

PREUVE. Ceci découle directement du fait que deux constantes c_i et c_j sont différentes. □

Proposition 6.4.2.22 *Les Règles [5.26] et [5.27] sont correctes et complètes.*

PREUVE. La correction de la Règle [5.26] est triviale.
Pour sa complétude, soit $\sigma \in \big|\!\big[\, B \wedge (X_i = u)^\bullet \wedge (X_i = v)^\bullet \,\big]\!\big|^e_\tau$.
Ainsi, $X_i\tau\sigma = \overline{u}^e\tau\sigma$ and $X_i\tau\sigma = \overline{v}^e\tau\sigma$. Par conséquent, $\sigma \in \big|\!\big[\, B \wedge (X_i = u)^\bullet \wedge (X_i = v)^\bullet \wedge u = v \,\big]\!\big|^e_\tau$ vu que $\overline{u}^e\tau\sigma = \overline{v}^e\tau\sigma$. La preuve de la correction et la complétude de la Règle [5.27] est similaire à celle de la Règle [5.26]. □

Proposition 6.4.2.23 *Les Règles [5.28], [5.29] et [5.30] sont correctes et complètes.*

PREUVE. Nous nous focalisons tout d'abord sur la Règle [5.28]. Selon la sémantique, nous avons ;

$$\begin{array}{ll}
\sigma \in \big|\!\big[\, (X_i = u)^\bullet \wedge X_i \in Forge_c(E', \mathcal{K}) \,\big]\!\big|^e_\tau & \text{ssi} \\
\sigma \in \big|\!\big[\, (X_i = u)^\bullet \,\big]\!\big|^e_\tau \cap [\![\, X_i \in Forge_c(E', \mathcal{K}) \,]\!]^e_\tau & \text{ssi} \\
X_i\tau\sigma = \overline{u}^e\tau\sigma \text{ and } X_i\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma) & \text{ssi} \\
X_i\tau\sigma = \overline{u}^e\tau\sigma \text{ and } \overline{u}^e\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma) & \text{ssi} \\
\sigma \in \big|\!\big[\, (X_i = u)^\bullet \wedge u \in Forge_c(E', \mathcal{K}) \,\big]\!\big|^e_\tau. &
\end{array}$$

Ensuite, la preuve de la correction et de la complétude de la Règle [5.29] est similaire à celle de la Règle [5.28]. En outre, celle de la Règle [5.30] est triviale vu que $E \subset E'$ et que $A\tau\sigma \in Dy_c(\overline{E}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, alors $A\tau\sigma \in Dy_c(\overline{E'}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. □

Proposition 6.4.2.24 *La Règle [5.31] est correcte et complète.*

PREUVE. Correction : Soit $\sigma \in [\![(X = w)^{sm} \wedge t \in Sub(w, E', \mathcal{E}, \mathcal{K}) \wedge X \notin Forge_c(E, \mathcal{K})]\!]^e_\tau$. Soit $u = X$, $\delta = \emptyset$, $\tau' = \tau$. Ainsi, $u\sigma = X\sigma = \overline{w}^e\delta\tau'\sigma$, ce qui conduit à $u\sigma \notin Dy_c(\overline{E'}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, et $\sigma \in [\![t \in Sub(w\delta, E', \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$. En outre, $u <^{\emptyset}_{\emptyset} X$ and $\emptyset \subseteq Dy(\overline{E'}^e\sigma, \emptyset)$, et par conséquent, $\sigma \in [\![t \in Sub_d(X, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$ selon la définition.

Complétude : soit $\sigma \in [\![(X = w)^{sm} \wedge t \in Sub_d(X, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$ où $(X = w)^{sm} \in \mathcal{E}$. Soient u, F, L les objets définis par $u \leqslant^L_F X$, $F\sigma \cap \overline{\mathcal{K}}^e\tau\sigma = \emptyset$ et $F\sigma \subseteq Dy(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\sigma)$ dans la définition de $t \in Sub_d(X, E', \mathcal{E}, \mathcal{K})$. Vu que Sub_d assure que $u <^L_F X$ dans le cas où $u\sigma = \overline{t}^e\tau\sigma$ et vu que $u <^L_F X$ est impossible, alors nous avons $u\sigma \neq \overline{t}^e\tau\sigma$ et $u = X$. Ainsi, u et F valident le deuxième cas de la sémantique du Sub avec $u \in \mathcal{X}$, et par conséquent, $\exists v, \delta, k, \tau'$ tels que $(X = v) \in \mathcal{E}$, $\delta = \emptyset$, $\tau' = \tau$, $k, i \notin \{t, X, \mathcal{E}\}$, $X \notin Forge_c(E, \mathcal{K})$, $u\sigma = \overline{v}^e\delta\tau'\sigma$ et $\sigma \in [\![t \in Sub(v\delta, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$. Cependant, vu que $(X = u)^{sm} \in \mathcal{E}$, nous choisissons $v = w$, ce qui conduit à $\sigma \in [\![(X = w)^{sm} \wedge t \in Sub(w, E', \mathcal{E}, \mathcal{K}) \wedge X \notin Forge_c(E, \mathcal{K})]\!]^e_\tau$. □

Proposition 6.4.2.25 *La Règle [5.32] est correcte et complète.*

PREUVE. La correction de la Règle [5.32] est triviale. Nous nous focalisons donc sur la complétude de cette règle.
Soit $\sigma \in [\![A \in Forge_c(E, \mathcal{K}) \wedge t \in Sub_d(A, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$.
Soient u, F tels que $u \leqslant^L_F A$ et $F\sigma \subseteq Dy(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\sigma)$.
Vu que Sub_d assure que $u <^L_F A$ dans la cas où $u\sigma = \overline{t}^e\tau\sigma$ et vu que $u <^L_F A$ est impossible, alors nous avons $u\sigma \neq \overline{t}^e\tau\sigma$ et $u = A$. Ainsi, u et F valident le deuxième cas de la sémantique du Sub. Par conséquent, $\exists v, \delta, k, \tau'$ tels que $u\sigma \notin Dy_c(\overline{E'}^e\tau'\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$. Néanmoins, $u\sigma = A\sigma$. Nous avons donc $A\sigma \notin Dy_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$. En outre, $A\sigma \in Dy_c(\overline{E}^e\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$. Cependant, $E \subseteq E'$. En effet, selon l'Invariant 6.4.1.3, $\mathcal{L}(A) \subseteq E'$. Ainsi, soit $\mathcal{L}(A) = E$ ou $\mathcal{L}(A) \subseteq E$. Pour le dernier cas, nous distinguons deux possibilités. Dans le premier cas, $A \in Forge_c(E, \mathcal{K})$ serait simplifiée en $A \in Forge_c(\mathcal{L}(A), \mathcal{K})$ si, au niveau $\mathcal{L}(A)$ nous avions $A \in Forge_c(\mathcal{L}(A), \mathcal{K})$ comme contrainte pour A. Dans le deuxième cas, $A \in Forge_c(E, \mathcal{K})$ serait éliminée si au niveau $\mathcal{L}(A)$, nous avions $A = u$ comme contrainte pour A. Ceci est dû au fait que, pour le niveau $\mathcal{L}(A)$, nous avons une contrainte de type $Forge$ ou d'égalité selon la Proposition 6.4.1.7. Nous concluons que $E \subseteq E'$. Par conséquent, $A\sigma \notin Dy_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$ et $A\sigma \in Dy_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\tau'\sigma)$, prouvant que $\sigma \in [\![\bot]\!]^e_\tau$. □

Proposition 6.4.2.26 *La Règle [5.33] est correcte et complète.*

PREUVE. Complétude : si $\sigma \in [\![t \in Sub_d(X_m, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$, alors $\exists (X_j = v) \in \mathcal{E}$, $\exists k \notin \{t, X_m, \mathcal{E}\}$, $\exists n_k$ t.q. $\tau' = \tau.[k \leftarrow n_k]$, $\sigma \in [\![t \in Sub(v\delta, E', \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$, $\sigma \in [\![X_m = v\delta]\!]^e_{\tau'}$
et $\sigma \notin [\![X_m \in Forge_c(E', \mathcal{K})]\!]^e_{\tau'}$, avec $\delta = \delta^k_{j,m}$. Pour prouver cela, soient u, F, L tels que $u \leqslant^L_F X_m\tau$, $F\sigma \cap \overline{\mathcal{K}}^e\tau\sigma = \emptyset$ et $F\sigma \subseteq Forge(\overline{E'}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$ à partir de la sémantique du Sub_d prouvant que $\sigma \in [\![t \in Sub_d(X_m, E', \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$. Notons que Sub_d est plus restrictive que le Sub dans le sens où elle assure que $u <^L_F X_m\tau$ dans le cas où $u\sigma = \overline{t}^e\tau\sigma$. Cependant, $u <^L_F X_m\tau$ est impossible, et donc $u\sigma \neq \overline{t}^e\tau\sigma$ et $u = X_m\tau$ nécessairement. Cela signifie que u, F valident le second cas de la sémantique du Sub, et par conséquent, $\exists v, \delta, k, i, j, \tau'$ t.q. $k, i \notin \{t, X_m, \mathcal{E}\}$, $\mathcal{D}om(\tau') = \mathcal{D}om(\tau) \cup \{k, i\}$, $u = X_i\tau'$, $(X_j = v) \in \mathcal{E}$, $\delta = \delta^k_{j,i}$, $u\sigma \notin Forge_c(\overline{E'}^e\tau\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, $u\sigma = \overline{v}^e\delta\tau'\sigma$, et $\sigma \in [\![t \in Sub(v\delta, E', \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$. Nous remarquons que $\tau(m) = \tau'(i)$ vu que $X_m\tau = u = X_i\tau'$, et donc la proposition suit en remplaçant i par m.

Correction : si $\exists (X_j = v) \in \mathcal{E}$, $\exists k \notin \{t, X_m, \mathcal{E}\}$, $\exists n_k$ t.q. $\tau" = \tau.[k \leftarrow n_k]$,
$\sigma \in [\![t \in Sub(v\delta, E', \mathcal{E}, \mathcal{K})]\!]^e_{\tau"}$, $\sigma \in [\![X_m = v\delta]\!]^e_{\tau"}$ et $\sigma \notin [\![X_m \in Forge_c(E', \mathcal{K})]\!]^e_{\tau"}$, avec $\delta = \delta^k_{j,m}$,

alors $\sigma \in [\![t \in Sub_d(X_m, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$. Supposons que la pré-condition ci-dessus est vraie pour les objets $X_j, v, \mathcal{E}, k, \tau" = \tau.[k \leftarrow n_k]$. Soit $u = X_m\tau$ et donc $u\sigma = X_m\tau"\sigma$. Également, soient i un indice frais, $\delta' = \delta^k_{j,i}$, et $\tau' = \tau".[i \leftarrow \tau(m)]$. Nous avons $u = X_i\tau'$, $(X_j = v) \in \mathcal{E}$, $k, i \notin \{t, X_m, \mathcal{E}\}$, $u\sigma = \overline{v}^e\delta'\tau'\sigma$, $u\sigma \notin Forge_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\sigma)$ et $\sigma \in [\![t \in Sub(v\delta', E', \mathcal{E}, \mathcal{K})]\!]^e_{\tau'}$. En outre, nous avons $u \leqslant^{\emptyset}_{\emptyset} X_m\tau$ et $\emptyset \subseteq Forge(\overline{E}^e\sigma, \overline{\mathcal{K}}^e\sigma)$, ce qui prouve que $\sigma \in [\![t \in Sub_d(X_m, E', \mathcal{E}, \mathcal{K})]\!]^e_\tau$ selon la définition. □

Proposition 6.4.2.27 *La Règle [5.34] est correcte et complète.*

Preuve. Nous désignons par *Rhs* la partie droite de la Règle [5.34]. Nous prouvons tout d'abord la complétude. Pour cela, supposons que $\sigma \in [\![X_m \in Forge_c(E', \mathcal{K})]\!]^e_\tau$, et que nous avons $\mathcal{M}(\overrightarrow{X})$ défini comme dans la Règle [5.34]. Nous distinguons deux cas :

- Soit $\exists o = 1..p\ \exists n_{k_o}$ t.q. $\sigma \in [\![X_m = u_o\delta_o]\!]^e_{\tau'}$ avec $\tau' = \tau.[k'_o \leftarrow n_{k_o}]$. Nous avons donc $\overline{u_o}^e\delta_o\tau'\sigma = X_m\tau'\sigma$, ce qui mène à $\overline{u_o}^e\delta_o\tau'\sigma \in Dy_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\sigma)$ et par conséquent, $\sigma \in [\![Rhs]\!]^e_\tau$;
- Soit il n'existe aucun o, i.e. $\forall o = 1..p, \forall n_{k_o}, \sigma \notin [\![X_m = u_o\delta_o]\!]^e_{\tau'}$ avec $\tau' = \tau.[k'_o \leftarrow n_{k_o}]$ i.e. $\overline{u_o}^e\delta_o\tau'\sigma \neq X_m\tau'\sigma$. Nous avons donc $\sigma \in \left|\!\left[\bigwedge_{o=1..p} \forall k'_o\, X_m \neq u_o\delta_o \right]\!\right|^e_\tau$. Ainsi, $\sigma \in [\![Rhs]\!]^e_\tau$;

Nous prouvons maintenant la correction de la Règle [5.34]. Pour cela, supposons que $\mathcal{M}(\overrightarrow{X})$ est défini comme dans la Règle [5.34]. Supposons aussi que $\sigma \in [\![Rhs]\!]^e_\tau$ puisque la seconde partie de *Rhs* est validée (sinon, la proposition est triviale). Ceci signifie que $\exists o = 1..p$, $\exists n_{k_o}$ t.q. $\overline{u_o}^e\delta_o\tau' \in Dy_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\sigma)$ et $X_m\tau'\sigma = \overline{u_o}^e\delta_o\tau'\sigma$ avec $\tau' = \tau.[k'_o \leftarrow n_{k_o}]$. Il suit que $X_m\tau\sigma \in Dy_c(\overline{E'}^e\sigma, \overline{\mathcal{K}}^e\sigma)$ vu que $X_m\tau = X_m\tau'$. □

Proposition 6.4.2.28 *La Règle [5.35] est correcte et complète.*

Preuve. La correction de la Règle [5.35] est triviale. Nous prouvons maintenant la complétude de cette règle. Pour cela, supposons que $\sigma \in [\![X_m = v]\!]^e_\tau$, i.e. $X_m\tau\sigma = \overline{v}^e\tau\sigma$, et $\mathcal{M}(\overrightarrow{X})$ selon les définitions de la Règle [5.35]. Notons $F = (\forall Q \exists R\, B_1 \vee .. \vee B_s)$ toute la formule dans laquelle la Règle [5.35] est utilisée, et supposons que $\sigma \in [\![F]\!]^e$. Nous désignons par *Rhs* la partie droite de la Règle [5.35]. Nous distinguons donc deux cas :

- Soit $\exists o = 1..p\ \exists n_{k_o}$ t.q. $\sigma \in [\![X_m = u_o\delta_o]\!]^e_{\tau'}$ avec $\tau' = \tau.[k'_o \leftarrow n_{k_o}]$. Nous avons donc $\overline{u_o}^e\delta_o\tau'\sigma = \overline{v}^e\tau'\sigma$, ce qui mène à $\sigma \in [\![Rhs]\!]^e_\tau$;
- Ou il n'existe aucun o, i.e. $\forall o = 1..p, \forall n_{k_o}, \sigma \notin [\![X_m = u_o\delta_o]\!]^e_{\tau'}$ avec $\tau' = \tau.[k'_o \leftarrow n_{k_o}]$ i.e. $\overline{u_o}^e\delta_o\tau'\sigma \neq X_m\tau'\sigma$, et donc $\sigma \in \left|\!\left[\bigwedge_{o=1..p} \forall k'_o\, X_m \neq u_o\delta_o \right]\!\right|^e_\tau$. En outre, selon le Corollaire 6.4.1.9, nous savons que, pour tout bloc de $B_1 \vee .. \vee B_s$, il existe une contrainte maître pour $\overrightarrow{X}$ dans ce bloc. De plus, $\exists \tau"$ avec $\forall i \in Q\ \tau"(i) = \tau'(m)$ tel que $\exists j\ \sigma \in [\![B_j]\!]^e_{\tau"}$. σ valide donc au moins une des contraintes maîtres pour $\overrightarrow{X}$ pour $\tau"$. Cependant, par hypothèse $\forall o = 1..p, \sigma \notin [\![X_m \neq u_o\delta_o]\!]^e_{\tau"}$. Ainsi, $\exists r = 1..q$ tel que $\sigma \in [\![X_{j_r} \in Forge_c(E'_r, \mathcal{K}_r)]\!]^e_{\tau"}$. Vu que $\tau"(j_r) = \tau'(m) = \tau(m)$, il suit que $\sigma \in [\![Rhs]\!]^e_\tau$.

□

6.5 Notre modèle est une extension des modèles classiques

Le but de cette section est de prouver la terminaison pour notre système d'inférence défini dans la Section 5.8 du Chapitre 5 et ceci pour les protocoles sans variables d'indice et sans *mpair*s générant des indices. Notons que pour ces protocoles, un système de contraintes ne contiendrait ni quantificateurs, ni variables d'indices et par conséquent pas de contraintes maîtres. Ce

résultat permet de justifier le fait que notre modèle est une extension des modèles classiques de vérification de protocoles cryptographiques. Nous commençons tout d'abord par définir dans la Section 6.5.1 le poids d'un système de contraintes en fonction de ses ingrédients, à savoir les blocs de contraintes, les contraintes élémentaires et les termes. Nous montrerons par la suite en Section 6.5.2 que, pour chacune des règles de notre système d'inférence, le poids du système de contraintes initial est supérieur à celui du système de contraintes résultant.

6.5.1 Poids des termes, contraintes élémentaires, blocs et systèmes de contraintes

Nous définissons tout d'abord le poids $\|\_\|$ des termes, des contraintes élémentaires, des blocs de contraintes et des systèmes de contraintes. Afin de définir le poids d'un terme, nous avons besoin d'introduire quelques définitions telles que la notion de rang d'un terme (Définition 6.5.1.1) et la notion de taille d'un terme (Définition 6.5.1.2).

Définition 6.5.1.1 *Rang d'un terme.*
Le rang d'un terme t est défini comme suit :
- $r(X) = max\{l \mid X \sqsubset_l Y, Y \in \mathcal{X}\}$ *pour* $X \in \mathcal{X}$
- $r(t) = max\{r(Y) \mid Y < t, Y \in \mathcal{X}\}$

Définition 6.5.1.2 *Taille d'un terme.*
Nous définissons la taille d'un terme t, notée |t|, comme suit :
- $|t| = 1$ *pour* $t \in \mathcal{X} \cup \mathcal{C}$
- $|f(u_1, .., u_m)| = 1 + |u_1| + .. + |u_m|$
- $|h(u)| = 2 + |u|$ *pour* $h \in H$

Nous étendons cette définition aux ensembles de termes : $|E| = \Sigma_{t \in E}|t|$ *pour* $E \subset T$.

Nous pouvons maintenant définir les poids des termes, des contraintes élémentaires et finalement des blocs de contraintes et des systèmes de contraintes. Nous commençons par définir le poids d'un terme :

Définition 6.5.1.3 *Poids d'un terme.*
Soit p la taille du protocole, i.e. la somme des tailles des messages. Nous définissons le poids d'un terme t, noté ||t||, comme suit :
- $\|X\| = p^{r(X)+1}$ *pour* $X \in \mathcal{X}$
- $\|c\| = 1$ *pour* $c \in \mathcal{C}$
- $\|f(u_1, .., u_m)\| = 1 + \|u_1\| + .. + \|u_m\|$
- $\|h(u)\| = 2 + \|u\|$ *pour* $h \in H$

Nous étendons cette définition d'une manière naturelle aux ensembles de termes : $\|E\| = \Sigma_{t \in E}\|t\|$ *pour* $E \subset T$.

Ensuite, le poids d'une contrainte élémentaire est donnée par la Définition 6.5.1.4.

Définition 6.5.1.4 *Poids d'une contrainte élémentaire.*
Soit st le nombre de sous-termes du protocole. Nous définissons le poids d'une contrainte élémentaire ctr, noté ||ctr||, comme suit :
- $\|t \in Forge(E, \mathcal{K})\| = \langle st - \#\mathcal{K}, \|t\| + \|E\| + |E| + 1\rangle$
- $\|t \in Forge_c(E, \mathcal{K})\| = \langle st - \#\mathcal{K}, \|t\| + \|E\| + |E|\rangle$
- $\|t \in Sub(w, E, \mathcal{E}, \mathcal{K})\| = \langle st - \#\mathcal{K}, \|t\| + \|w\| + |E| + 1\rangle$
- $\|t \in Sub_d(w, E, \mathcal{E}, \mathcal{K})\| = \langle st - \#\mathcal{K}, \|t\| + \|w\| + |E|\rangle$

– $\|t = u\| = \langle 0, \|t\| + \|u\| \rangle$

En outre, un bloc de contraintes est constitué de plusieurs contraintes élémentaires. Le poids d'un bloc est alors donné par la Définition 6.5.1.5.

Définition 6.5.1.5 *Poids d'un bloc de contraintes.*

Soient NcE le nombre maximal de contraintes d'égalité de la forme $X = u$ où $X \in X$ et NcN le nombre maximal de contraintes négatives de type $Forge$. Considérons un bloc de contraintes $B = ctr_1 \wedge ..ctr_l$. Nous désignons par Nc_B le nombre de contraintes négatives dans B, Sc_B le nombre de contraintes sous-maîtres dans B et Ec_B le nombre de contraintes d'égalité de la forme $X = u$ ($X \in \mathcal{X}$) dans B. Ensuite, nous notons $\langle\rangle$ l'ordre lexicographique et $[]$ le multi-ensemble.

Nous définissons alors le poids d'un bloc B comme suit :

$$\|B\| = \left\langle NcN - Nc_B,\ NcE - Sc_B,\ [\|ctr_i\|]_{ctr_i \in B},\ NcE - Ec_B \right\rangle$$

Finalement, vu qu'un système de contraintes est constitué de plusieurs blocs de contraintes, nous pouvons maintenant définir le poids d'un tel système de contraintes :

Définition 6.5.1.6 *Poids d'un système de contraintes.*

Considérons un système de contraintes : $S = B_1 \wedge ..B_l$. Nous définissons le poids de S comme suit :

$$\|S\| = [\|B_i\|]_{B_i \in S}$$

6.5.2 Terminaison pour les protocoles sans indices et sans *mpair*

Le but de cette section est de prouver la Proposition 6.2.0.6 de la Section 6.2 :

Proposition 6.2.0.6 *Terminaison pour les protocoles sans indices et sans $mpair(,)$.*

L'algorithme 6.2 termine pour les protocoles sans variables d'indices et sans $mpair(,)$.

Preuve. Nous prouvons la Proposition 6.2.0.6 en montrant que chaque règle de notre système d'inférence défini en Section 5.8 du Chapitre 5 fait diminuer le poids de notre système de contraintes S, i.e. considérant une règle r, $\|\text{post}(r)\| < \|\text{pre}(r)\|$. Tout d'abord, notons que les règles de notre système d'inférence n'éliminent pas de contraintes négatives. Ainsi, $\forall B \in S$, $NcN - Nc_B$ reste inchangé ou bien diminue pour le cas de la Règle [5.31]. Par conséquent, la Règle [5.31] fait diminuer $\|S\|$. Les Règles [5.3], [5.14], [5.20], [5.22] et [5.32] éliminent un bloc du système de contraintes vu qu'elles mènent toutes à $\bot$. Elles font donc diminuer $\|S\|$. La Règle [5.22] élimine une contrainte d'un certain bloc de S vu qu'elle mène à $\top$. Elle diminue alors $\|S\|$. La Règle [5.1] change une contrainte d'égalité $ctr \in B$ en une autre de la forme $(X = u)$ où $X \in X$. Ainsi, le nombre de contraintes sous-maîtres et le poids de contraintes de B demeurent inchangés. En outre, $NcE - Ec_B$ diminue vu que Ec_B augmente. Par conséquent, $\|B\|$ diminue et ainsi fait $\|S\|$.

La Règle [5.8] ajoute une contrainte sous-maître à un bloc $B \in S$. Ainsi, $\|S\|$ diminue.

Pour la Règle [5.2], vu que $(X = u)^{sm} \in B$, alors $\|X\| > \|u\|$.

En effet, $\|u\| \leq |u| * p^{max\{\text{r}(Y) | Y < u, Y \in \mathcal{X}\}} \leq p^{\text{r}(X)+1}$. Ensuite, $\|Y = X\| = \|Y\| + \|X\| > \|Y\| + \|u\| = \|Y = u\|$. En outre, le nombre de contraintes sous-maîtres reste inchangé. Ainsi, $\|S\|$ diminue.

Pour la Règle [5.9], nous avons $\|\text{pre}(R\ [5.9])\| > \|\text{post}(R\ [5.9])\|$. En effet, $\|\text{pre}(R\ [5.9])\| = \langle k, \|t\| + \|E\| + |E| + 1\rangle$ et $\|\text{post}(R\ [5.9])\| = [\langle k, \|t\| + \|E\| + |E|\rangle, \langle k, \|t\| + \|w\| + |E| + 1\rangle]$, où $k = st - \#\mathcal{K}$ et $w \in E$. En outre, la Règle [5.9] ne change pas le nombre de contraintes

sous-maîtres. Ainsi, $\|S\|$ diminue.
Pour la Règle [5.10], nous avons $\|\text{pre}(R\ [5.10])\| = \langle k, 1 + \|t_1\| + .. + \|t_m\| + \|E\| + |E|\rangle$ et $\|\text{post}(R\ [5.10])\| = [\langle k, \|t_i\| + \|E\| + |E| + 1\rangle]_{t_i < \langle t_1 .. t_m \rangle}$, où $k = st - \#\mathcal{K}$. En outre, la Règle [5.10] n'ajoute et n'élimine aucune contrainte sous-maître. Ensuite, $\|\text{pre}(R\ [5.10])\| > \|\text{post}(R\ [5.10])\|$. Ainsi, $\|B\|$ et $\|S\|$ diminuent.
Nous montrons d'une manière similaire à la Règle [5.10] que pour la Règle [5.11], $\|S\|$ diminue.
Pour la Règle [5.12], $\|\text{pre}(R\ [5.12])\| > \|\text{post}(R\ [5.12])\|$.
En effet, $\|\text{pre}(R\ [5.12])\| = \langle k, 2 + \|t\| + \|E\| + |E|\rangle$ et $\|\text{post}(R\ [5.12])\| = \langle k, 1 + \|t\| + \|E\| + |E|\rangle$, où $k = st - \#\mathcal{K}$. En outre, la Règle [5.12] ne change pas le nombre de contraintes sous-maîtres. Ainsi, $\|B\|$ et $\|S\|$ diminuent.
Pour la Règle [5.15], $\|\text{pre}(R\ [5.15])\| = \langle k, 1 + \|t\| + \|u\| + |E|\rangle$. Ensuite, nous distinguons deux cas pour post(R [5.15]). Dans le premier cas, $\|\text{post}(R\ [5.15])\| = \langle k, \|t\| + \|u\|\rangle$. Dans le deuxième cas, $\|\text{post}(R\ [5.15])\| = [\langle k, \|t\| + \|u\|\rangle, \langle k, \|t\| + \|u\| + |E|\rangle]$. Dans les deux cas, $\|\text{pre}(R\ [5.15])\| > \|\text{post}(R\ [5.15])\|$. En outre, vu que cette règle ne modifie pas le nombre de contraintes sous-maîtres, $\|S\|$ diminue.
Nous montrons d'une manière similaire à la Règle [5.10] que, pour la Règle [5.16], $\|S\|$ diminue.
Pour la Règle [5.17], tout d'abord, $\|\text{pre}(R\ [5.17])\| = \langle k, 1 + \|t\| + \|u\| + \|b\| + |E|\rangle$ où $k = st - \#\mathcal{K}$. Ensuite, $\|\text{post}(R\ [5.17])\| = [\langle k, 1 + \|t\| + \|u\| + |E|\rangle, \langle k', 1 + \|b\| + \|E\| + |E|\rangle]$ où $k' = st - \#\mathcal{K} \cup \{\{u\}_p^p\}$. Vu que $k' < k$, $\|\text{pre}(R\ [5.17])\| > \|\text{post}(R\ [5.17])\|$.
Nous montrons d'une manière similaire à la Règle [5.17] que, pour la Règle [5.18], $\|S\|$ diminue.
Pour la Règle [5.23], $\|\text{pre}(R\ [5.23])\| = \langle k, 2 + \|u_1\| + .. + \|u_m\| + \|w_1\| + .. + \|w_m\|\rangle$. Ensuite, $\|\text{pre}(R\ [5.23])\| = [\langle k, \|u_i\| + \|w_i\|\rangle]_{i=1..m}$ où $k = st - \#\mathcal{K}$.
Ainsi, $\|\text{pre}(R\ [5.23])\| > \|\text{post}(R\ [5.23])\|$ et vu que le nombre de contraintes sous-maîtres reste inchangé, $\|S\|$ diminue.
Nous montrons d'une manière similaire à la Règle [5.2] que, pour la Règle [5.27], $\|S\|$ diminue vu que $\|X\| > \|u\|$ et par la suite, $\|X = v\| > \|u = v\|$.
Nous montrons d'une manière similaire à la Règle [5.27] que, pour la Règle [5.29], $\|S\|$ diminue vu que $\|X\| > \|u\|$ et par la suite, $\|X \in Forge_c(E', \mathcal{K})\| > \|u \in Forge_c(E', \mathcal{K})\|$.
La Règle [5.30] élimine une contrainte d'un bloc. Ainsi, $\|S\|$ diminue. □

6.6 Terminaison pour les protocoles bien tagués avec clefs autonomes

Le but de cette section est de prouver la terminaison de notre système d'inférence défini en Section 5.8 du Chapitre 5. Ce résultat a été énoncé en Section 6.2 :

Lemme 6.2.0.8 *Terminaison de la normalisation.*
L'Algorithme 6.2 termine pour les protocoles bien tagués avec clefs autonomes.

La preuve de ce lemme est effectué en deux grandes parties :

Première étape : borner l'ensemble d'indices possibles qu'on peut générer. Nous bornons tout d'abord l'ensemble d'indices quantifiés *universellement*. Ensuite, nous prouvons que l'ensemble d'indices *existentiels* est fini. Nous prouvons ce résultat d'une manière incrémentale en considérant des sous-ensembles de contraintes.

Deuxième étape : définir un poids pour notre système de contraintes qui diminue pour l'application de chaque règle de notre système d'inférence. Pour la définition du poids des ingrédients de notre système de contraintes, nous nous inspirons des poids définis pour les protocoles sans

indices et sans *mpair*. D'ailleurs, nous conservons la même définition des poids des contraintes élémentaires (Définition 6.5.1.4) et des systèmes de contraintes (Définition 6.5.1.6). Le poids d'un terme dans le bloc est défini comme étant la somme des poids de ses symboles. Les opérateurs de hachage et de *mpair* ont 2 comme poids. Les autres fonctions ont 1 comme poids. Le poids d'une variable X est défini afin de diminuer le poids du bloc contenant X quand on remplace X par sa valeur dans ce bloc (par exemple, remplacer $X = u$ par $v = u$ quand $X = v$ appartient à ce bloc). Nous rappelons aussi que le poids d'une contrainte élémentaire est défini comme étant l'ordre lexicographique appliqué au nombre de clefs qui ne sont pas dans $\mathcal{K}$ en premier, et à une mesure numérique sur les paramètres des contraintes en second. Cette mesure est définie de telle manière que le poids des contraintes $Forge$ est supérieur à celui des contraintes $Forge_c$ et des contraintes Sub. De la même manière, le poids des contraintes Sub est supérieur à celui des contraintes Sub_d et des contraintes d'égalité. Puisque, d'après la première étape, nous avons borné l'ensemble d'indices susceptibles d'être générés par notre système de contraintes, nous avons une borne du nombre de contraintes Forge$(X \in Forge_c(E, \mathcal{K}))$, ou $(X = u)$, ou encore de contraintes négative où $X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$. Le poids d'un système de contraintes est défini comme étant l'ordre lexicographique qui *diminue* quand :

1. le nombre de remplacement des variables indicées dans l'historique du bloc augmente,
2. une contrainte *finale* apparaît dans le bloc,
3. une contrainte *négative* apparaît dans le bloc,
4. une contrainte *sous-maître* est générée dans le bloc,
5. une contrainte *maître* de type *égalité* est générée dans le bloc,
6. une contrainte *maître* de type *Forge* est générée dans le bloc,
7. le *multi-ensemble* des poids des contraintes du bloc diminue,
8. une contrainte de la forme $X = u$ où $X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}$ est générée dans le bloc.

Notons que la liste de ces conditions est donnée dans un ordre lexicographique. Pour exemple, le poids du système de contraintes diminue si une contrainte négative apparaît dans le bloc *et* le nombre de contraintes finales reste inchangé. Le poids d'un système de contraintes, tel qu'il est défini en Définition 6.5.1.6, est défini comme étant le multi-ensemble des poids des différents blocs.

Dans cette section, nous commençons par relever quelques propriétés de notre système de contraintes liées aux variables d'indices en prouvant quelques petits invariants en Section 6.6.1. Ces invariants seront utilisés essentiellement pour prouver la première partie de la preuve de terminaison. Cette première partie de la preuve de terminaison, élaborée en Section 6.6.2, a pour but de montrer que l'ensemble d'indices généré par notre système de contraintes est borné. Une fois que l'ensemble d'indices que notre système de contraintes peut générer est fixé, nous prouvons la terminaison de notre système de règles pour les protocoles bien tagués avec clefs autonomes en Section 6.6.3. En effet, nous définissons le poids des termes, des contraintes élémentaires, des blocs et des systèmes de contraintes en s'inspirant des poids définis en Section 6.5 pour les protocoles sans indices et sans *mpair*s.

6.6.1 Propriétés de notre système d'inférence liées aux indices

Nous présentons dans cette section, quelques propriétés de notre système de contraintes liées aux indices et plus spécialement aux conséquences de taguage. Ces propriétés sont représentées sous forme de quatre invariants qui seront utilisés spécialement pour la preuve de l'existence

d'une borne pour l'ensemble des indices susceptibles d'être générés et par conséquence, pour la preuve de terminaison de notre système d'inférence.

Nous commençons par la premier invariant (Invariant 6.6.1.1) qui décrit une conséquence directe du taguage. Cet invariant déclare que, pour une contrainte de type Sub, si le terme t' à l'intérieur du Sub contient une variable indicée, alors tout sous-terme de t' contenant cette variable indicée sans traverser de $mpair$ est tagué. De la même manière, pour une contrainte d'égalité contenant une variable indicée de chaque côté, l'une des deux variables est taguée. En plus, si les deux variables sont des sous-termes des deux termes de l'égalité sans traverser de $mpair$ alors il s'agit du même indice des deux côtés de l'égalité. L'invariant 6.6.1.1 sera utilisé pour la preuve des Invariants 6.6.1.2 et 6.6.1.3 et pour la preuve de la Proposition 6.6.2.7.

Invariant 6.6.1.1
Pour une contrainte $(t \in Sub(t', E, \mathcal{E}, \mathcal{K}))$ *où il existe* $X_i \leq t'$*, nous avons pour tout* $u \leq t'$ *tel que* $X_i \leqslant_m u$*,* u *est tagué.*
Pour une contrainte $(u = v)$ *où il existe* $X_i < u$*,* $Y_j < v$ *tels que* $X_i, Y_j \in \mathcal{X}_\mathcal{I}$*, nous avons* X_i *ou* Y_j *est tagué. En outre, si* $X_i \leqslant_m u$ *et* $Y_j \leqslant_m v$*, alors* $i = j$*.*

Preuve. Le premier groupe de règles manipule des contraintes d'égalité qui satisfont l'invariant. En effet, la Règle [5.3] élimine tout le bloc. Ainsi l'invariant reste valide. Les Règles [5.6] et [5.7] transforment des règles d'égalité et en génèrent de nouvelles. Néanmoins, ces contraintes ne suivent pas les formes des contraintes définies dans l'Invariant 6.6.1.1, i.e. ayant des variables indexées dans les deux termes composant l'égalité. Le reste des règles du groupe G_1 ne modifient pas de contraintes puisqu'elles traitent seulement l'étiquetage. L'invariant est donc valable.
Les règles du groupe G_2 traitent des contraintes de type $Forge$. Seule la Règle [5.9] génère une contrainte de type Sub : $(t \in Sub(w, E, \mathcal{E}, \mathcal{K}))$ avec $w \in E$. Cependant, selon la seconde condition de la définition des protocoles bien tagués (voir Définition 5.6.2.3), w est tagué. En outre, selon la troisième condition de la même définition, chaque sous-terme de w, ayant une variable indexée comme sous-terme sans traverser d'opérateurs $mpair$, est tagué. Ainsi, si $\exists X_i \leq t'$ alors $\forall u \leq t'$ tel que $X_i \leqslant_m u$, u est tagué, et donc, l'invariant suit. Les autres règles de ce groupe ne génèrent ni des contraintes d'égalité, ni des contraintes de type Sub. L'invariant reste donc valable.
La Règle [5.15] du groupe G_3 transforme une contrainte Sub en une autre contrainte Sub_d ou en une contrainte d'égalité. La contrainte Sub_d préserve le même terme u, et donc, l'invariant reste valide. La contrainte d'égalité est de la forme $(t = u)$ où u est le terme de la contrainte Sub. Néanmoins, par hypothèse d'induction, si $\exists X_i \leq u$ alors $\forall v \leq u$ tel que $X_i \leqslant_m v$, v est tagué. Ainsi, la première condition de l'invariant pour la contrainte $(t = u)$ est satisfaite. D'ailleurs, si $X_i \leqslant_m u$ et $Y_j \leqslant_m t$, alors u est tagué vu que X_i est tagué et par application de la troisième condition de la Définition 5.6.2.3. Par conséquent, t doit être tagué par le même tag que u et donc, $i = j$. Les autres règles du groupe G_3 transforment une contrainte Sub_d en de nouvelles contraintes Sub tout en décomposant le terme à l'intérieur de cette contrainte Sub_d. Cependant, si l'invariant est valide pour une contrainte $(t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$ pour un terme u, alors il reste valable pour un sous-terme u' de u $(u' \leq u)$. En effet, si $\exists X_i \leq u'$, alors, $X_i \leq u$. Par hypothèse d'induction, $\forall v \leq u$ tel que $X_i \leqslant_m v$, v est tagué. Par conséquent, $\forall v \leq u' \leq u$ tel que $X_i \leqslant_m v$, v est tagué. Nous concluons que toutes les règles du groupe G_3 préservent l'invariant.
Les règles du groupe G_4 manipulent des contraintes d'égalité. La Règle [5.21] élimine la contrainte traitée. La Règle [5.22] élimine tout le bloc. La Règle [5.25] remplace un indice par un autre dans tout le bloc. Ainsi, toutes ces règles satisfont l'invariant. La Règle [5.23] décompose une contrainte d'égalité de deux termes $(u = v)$ en une autre contrainte d'égalité de deux sous-termes

$(u' = v')$ avec $u' \leq u$ et $v' \leq v$. Ainsi, si $\exists X_i < u'$, $Y_j < v'$ alors $\exists X_i < u$, $Y_j < v$. Néanmoins, par hypothèse d'induction, X_i ou Y_j est tagué. En outre, si $X_i \leqslant_m u'$ et $Y_j \leqslant_m v'$ alors, $X_i \leqslant_m u$ et $Y_j \leqslant_m v$, et donc, $i = j$. La Règle [5.24] décompose une contrainte d'égalité de deux termes $(mpair(k,u) = mpair(l,w))$ en une autre contrainte d'égalité de deux sous-termes $(u = w\delta_{l,k})$. Si $\exists X_i < u < mpair(k,u)$, $Y_j < w\delta_{l,k} < mpair(k,w)$ tels que $X_i, Y_j \in \mathcal{X}_{\mathcal{I}}$, alors, X_i ou Y_j est tagué par hypothèse d'induction. En outre, $Var_{\mathcal{I}}(u) \subseteq \{k\}$ et $Var_{\mathcal{I}}(w\delta_{l,k}) \subseteq \{k\}$. Ainsi, si $X_i \leqslant_m u$ et $Y_j \leqslant_m w\delta_{l,k}$ alors $i = k = j$.
La Règle [5.26] du groupe G_5 génère une contrainte d'égalité $(u = v)$. Supposons que $Y_j < u$ et $Z_k < v$. Par hypothèse d'induction, pour la contrainte $(X_i = u)$, Y_j est tagué. D'une manière similaire, Z_k est tagué. Ainsi, la première condition de l'invariant est satisfaite. Supposons maintenant que $Y_j \leqslant_m u$ et $Z_k \leqslant_m v$. Par hypothèse d'induction, concernant la contrainte $(X_i = u)$, nous avons $i = j$. D'une manière semblable, pour la contrainte $(X_i = v)$, nous avons $i = k$. Ainsi, $j = k$. Par conséquent, la deuxième condition de l'invariant est valide. Pour la Règle [5.27], la contrainte d'égalité générée $(u = v)$ ne peut pas avoir $X_i \leq u$ ou $Y_j \leq v$ par priorité des Règles [5.6] et [5.7] remplaçant les variables indicées par des non-indicées. Pour les mêmes raisons, la Règle [5.31] ne peut pas générer de contrainte $(t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))$ ayant de variables indicées dans w. Les Règles [5.28], [5.29] et [5.30] ne modifient pas les contraintes d'égalité. La Règle [5.32] élimine tout le bloc. Par conséquent, l'invariant reste valable par application des règles du groupe G_5.
La Règle [5.33] du groupe G_6 génère une contrainte $(t \in Sub_d(u\delta, E, \mathcal{E}, \mathcal{K}))$. Cependant, u provient d'une contrainte maître : $(X_i = u)$. Par hypothèse d'induction, si $\exists Y_j < u$ alors Y_j est tagué. En plus, vu que u est un terme du protocole, selon la troisième condition de la Définition 5.6.2.3, $\forall u' \leq u$ tel que $Y_j \leqslant_m u'$, u' est tagué. Ainsi, la contrainte Sub générée valide l'invariant. La Règle [5.34] ne génère, ni de contraintes Sub, ni de contraintes d'égalité non finales. Ainsi, l'invariant est valable. La Règle [5.35] génère une contrainte d'égalité $(u_o\delta_o = v)$. Supposons que $\exists Y_j$ et Z_k tel que $Y_j < u_o\delta_o$ et $Z_k < v$. Par hypothèse d'induction, pour la contrainte $(X_m = v)$, Z_k est tagué. La première condition de l'invariant est alors satisfaite. Supposons maintenant que $Y_j \leqslant_m u_o\delta_o$ et $Z_k \leqslant_m v$. Par hypothèse d'induction, pour la contrainte $(X_m = v)$, nous avons $m = k$. Par hypothèse d'induction, pour la contrainte $(X_o = u_o)$, où $Y_l < u_o$ nous avons $o = l$. Ainsi, $Var_{\mathcal{I}}(u_o\delta_o) \subseteq \{m\}$, ce qui mène à $j = m$. Par conséquent, $j = k$ et l'invariant reste donc valable par application de la Règle [5.35]. □

Le deuxième invariant (Invariant 6.6.1.2) se focalise sur la manière dont les indices sont quantifiés dans les contraintes et plus spécialement ceux qui sont quantifiés universellement. Il exprime le fait que pour les contraintes $Forge$ ou Sub, si le terme à composer (dans les deux cas de $Forge$ ou Sub) contient une variable indicée sans traverser des $mpairs$ alors l'indice existant dans ce terme est quantifié universellement. De la même manière, pour une contrainte d'égalité dont les deux termes contiennent deux variables indicées sans traverser des $mpair$s, les deux indices de ces deux variables sont quantifiés universellement. Cet invariant sera utilisé pour la preuve de l'Invariant 6.6.1.3, des Propositions 6.6.2.7, 6.6.2.8 et 6.6.2.11 ainsi que pour la preuve du Lemme 6.6.2.14.

Invariant 6.6.1.2 *Pour une contrainte de la forme* $(t \in Forge_c(E, \mathcal{K}))$, $(t \in Forge(E, \mathcal{K}))$, $(t \in Sub(u, E, \mathcal{E}, \mathcal{K}))$, *ou* $(t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$, *où* $\exists X_i \leqslant_m t$, *nous avons* $Var_{\mathcal{I}}(t) \subset Q$.
Pour une contrainte non finale de la forme $u = v$ *où* $\exists Y_j$ *et* Z_k *t.q.* $Y_j \leqslant_m u$ *et* $Z_k \leqslant_m v$, *nous avons* $j, k \in Q$.

Preuve. Les règles du groupe G_1 ne génèrent pas de nouvelles contraintes de la forme $Forge$ ou Sub ou des contraintes d'égalité de la forme définie dans l'Invariant 6.6.1.2. L'invariant reste

donc valide.
La Règle [5.9] du groupe G_2 génère à partir d'une contrainte $Forge$, une contrainte de type $Forge_c$ ou de type Sub mais tout en gardant le même terme t dans les deux cas. Néanmoins, par hypothèse d'induction, $Var_{\mathcal{I}}(t) \subset Q$. Ainsi, l'invariant reste satisfait. Les Règles [5.10], [5.11] et [5.12] transforment une contrainte $t \in Forge_c(E, \mathcal{K})$ en une autre contrainte $(t' \in Forge(E, \mathcal{K}))$ avec $t' \leqslant_m t$. Or, $Var_{\mathcal{I}}(t') \subseteq Var_{\mathcal{I}}(t)$ et $Var_{\mathcal{I}}(t) \subset Q$. Ainsi, $Var_{\mathcal{I}}(t) \subset Q$, ce qui satisfait l'invariant. La Règle [5.13] génère une nouvelle contrainte $(t \in Forge(E, \mathcal{K}))$ où $Var_{\mathcal{I}}(t) \subset \{k\}$ et $k \in Q$. L'invariant est donc valide. La Règle [5.14] élimine tout le bloc. Nous concluons que le groupe G_2 satisfait l'invariant.
La Règle [5.15] transforme une contrainte Sub en une autre contrainte, soit de type Sub_d gardant le même terme t, soit de type égalité. Pour la nouvelle contrainte Sub_d, par hypothèse d'induction, nous avons $Var_{\mathcal{I}}(t) \subset Q$, ce qui satisfait l'invariant. Pour la contrainte d'égalité $(t = u)$, supposons que $\exists X_i, Y_j$ tels que $X_i \leqslant_m t$ et $Y_j \leqslant_m u$. Selon l'Invariant 6.6.1.1 appliqué à la contrainte $(t = u)$, nous avons $i = j$. Or, par hypothèse d'induction, $Var_{\mathcal{I}}(t) \subset Q$. Comme nous remplaçons les indices quantifiés universellement que par des indices quantifiés universellement, nous avons $i, j \in Q$. Les Règles [5.16] et [5.19] transforment une contrainte Sub en une autre contrainte tout en conservant le même terme t. Or, par hypothèse d'induction, $Var_{\mathcal{I}}(t) \subset Q$, et donc l'invariant reste valide. Les Règles [5.17] et [5.18] génèrent, à partir d'une contrainte Sub, à la fois une contrainte de type Sub_d utilisant le même terme t, et une nouvelle contrainte de type $Forge$ pour composer la clef b. Pour la nouvelle contrainte Sub_d, par hypothèse d'induction, nous avons $Var_{\mathcal{I}}(t) \subset Q$ qui satisfait l'invariant. Pour la contrainte $Forge$, b est une clef. Selon la restriction de clefs autonomes, nous avons $Var_{\mathcal{I}}(b) = \emptyset$, qui satisfait l'invariant. La Règle [5.20] élimine tout le bloc. Nous concluons que les règles du groupe G_3 satisfont l'invariant.
La Règle [5.21] du groupe G_4 élimine une contrainte. La Règle [5.22] élimine tout le bloc. La Règle [5.23] transforme une contrainte d'égalité $(u = v)$ en une autre contrainte $(u' = v')$ où $u' \leqslant_m u$ et $v' \leqslant_m v$. Ainsi, si $\exists X_i, Y_j$ tels que $X_i \leqslant_m u'$ et $Y_j \leqslant_m v'$, alors $X_i \leqslant_m u$ et $Y_j \leqslant_m v$. Or, par hypothèse d'induction, $i, j \in Q$, ce qui satisfait l'invariant. La Règle [5.21] transforme une contrainte d'égalité en une autre $(u = w\delta_{l,k})$ où $Var_{\mathcal{I}}(u) = Var_{\mathcal{I}}(v) \subseteq \{k\}$ et $k \in Q$, ce qui satisfait l'invariant. Par conséquent, l'application des règles du groupe G_4 préserve l'invariant.
La Règle [5.26] génère une nouvelle contrainte $(u = v)$. Supposons que $\exists Z_k, Y_j$ tel que $Z_k \leqslant_m u$ et $Y_j \leqslant_m v$. Par hypothèse d'induction, en considérant la contrainte $(X_i = u)$, nous avons $i, k \in Q$. De même, par hypothèse d'induction, en considérant la contrainte $(X_i = v)$, nous avons $i, j \in Q$. Ainsi, $j, k \in Q$, ce qui valide l'invariant. La Règle [5.27] génère une contrainte d'égalité $(u = v)$. Or, u et v proviennent des contraintes $(X = u)$ et $(X = v)$. Ainsi, $\nexists X_i$ tel que $X_i \leqslant_m u$ ou $X_i \leqslant_m v$ par priorité des Règles [5.6] et [5.7]. De même, la Règle [5.29] ne génère pas de contrainte $(u \in Forge_c(E, \mathcal{K}))$ avec $X_i \leqslant_m u$. Les Règles [5.27] et [5.29] satisfont donc l'invariant. La Règle [5.28] génère une nouvelle contrainte $(u \in Forge_c(E, \mathcal{K}))$. Supposons que $\exists Y_j$ tel que $Y_j \leqslant_m u$. Or, u provient d'une contrainte $X_i = u$. Selon l'Invariant 6.6.1.1, $i = j$. De plus, par hypothèse d'induction, nous avons $i \in Q$ et donc $j \in Q$, ce qui valide l'invariant. La Règle [5.30] ne génère pas de nouvelles contraintes. La Règle [5.31] génère une nouvelle contrainte Sub tout en conservant le même terme t. La Règle [5.32] élimine tout le bloc. Par conséquent, l'application des règles du groupe G_5 valide l'invariant.
La Règle [5.33] génère une nouvelle contrainte Sub mais en utilisant le même terme t. L'invariant reste donc valide. La Règle [5.34] génère une nouvelle contrainte $(u_o\delta_o \in Forge_c(E', \mathcal{K}))$. Supposons que $\exists Y_j$ tel que $Y_j \leqslant_m u_o\delta_o$. Or, selon l'Invariant 6.6.1.1 appliqué à la contrainte maître $(X_o = u_o)$ en tenant compte de δ_o, nous avons $j = m$. En outre, par hypothèse d'induction appliquée à la contrainte $(X_m \in Forge_c(E', \mathcal{K}))$, nous avons $m \in Q$, ce qui valide l'invariant. La Règle [5.35] génère deux nouvelles contraintes. La première est $(v \in Forge_c(E'_r, \mathcal{K}_r))$. Supposons

que $\exists Y_j$ tel que $Y_j \leqslant_m v$. Selon l'hypothèse d'induction, appliquée à la contrainte $(X_m = v)$, nous avons $m, j \in Q$, ce qui satisfait l'invariant. La deuxième contrainte générée est $(u_o\delta_o = v)$. Supposons que $\exists Y_j, Z_k$ tels que $Y_j \leqslant_m v$ et $Z_k \leqslant_m u_o\delta_o$. Or, par hypothèse d'induction, appliquée à la contrainte maître $(X_o = u_o)$, en tenant compte de δ_o et de la contrainte $(X_m = v)$, nous avons $m, j, k \in Q$, ce qui satisfait l'invariant. Nous concluons que l'application des règles du groupe G_6 valide l'invariant. □

La troisième propriété de notre système de contraintes concernant les indices exprime que, pour le cas d'une égalité de deux termes contenant d'un côté une variable indicée et de l'autre côté une constante indicée, alors ces deux indices ne peuvent pas être tous les deux quantifiés existentiellement.

Invariant 6.6.1.3 *Pour une contrainte $(u = v)$, où $X_i \leqslant_m u$ et $c_j \leqslant_m v$, alors i et j ne peuvent pas être quantifiés tous les deux existentiellement.*

Preuve. Les groupes G_1 et G_2 ne génèrent pas de contraintes d'égalité de la forme $(u = v)$ où $\exists X_i \in \mathcal{X}_\mathcal{I}$ et $c_j \in \mathcal{C}_\mathcal{I}$ tels que $X_i \leqslant_m u$ et $c_j \leqslant_m v$. Ainsi, l'invariant reste valide.
Pour le groupe G_3, seule la Règle [5.15] peut générer une contrainte d'égalité $(t = u)$, à partir d'une contrainte $(t \in Sub(u, E, \mathcal{E}, \mathcal{K}))$. Supposons que $\exists X_i \in \mathcal{X}_\mathcal{I}$ et $c_j \in \mathcal{C}_\mathcal{I}$ tels que $i, j \in R$ et, soit $(X_i \leqslant_m u$ et $c_j \leqslant_m t)$, soit $(X_i \leqslant_m t$ et $c_j \leqslant_m u)$. Dans le premier cas, i.e. $(X_i \leqslant_m u$ et $c_j \leqslant_m t)$, selon l'Invariant 6.6.1.1, X_i est tagué. Selon la troisième condition de la Définition 5.6.2.3, u est aussi tagué. Ainsi, v doit être tagué par le même tag que u et donc $i = j$, ce qui contredit l'hypothèse. Dans le deuxième cas, i.e. $X_i \leqslant_m t$ et $c_j \leqslant_m u$, selon l'Invariant 6.6.1.2, nous avons $i \in Q$ qui contredit l'hypothèse : $i \in R$. Ainsi, pour les deux cas, i et j ne peuvent pas être quantifiés existentiellement.
La Règle [5.21] élimine une contrainte. La Règle [5.22] élimine tout le bloc. La Règle [5.25] remplace un indice par un autre. Néanmoins, elle ne remplace jamais un indice quantifié universellement par un indice quantifié existentiellement. Ainsi, ces trois règles du groupe G_4 satisfont l'invariant. La Règle [5.23] décompose une contrainte d'égalité entre deux termes $(u = v)$ en une autre contrainte d'égalité entre deux sous-termes $((u' = v')$ avec $u' <_m u$ et $v' <_m v)$ sans générer d'indices. Par hypothèse d'induction, si $\exists X_i \in \mathcal{X}_\mathcal{I}$ et $c_j \in \mathcal{C}_\mathcal{I}$ tels que $X_i \leqslant_m u' <_m u$ et $c_j \leqslant_m v' <_m v$ alors i et j ne peuvent pas être tous les deux quantifiés existentiellement. La Règle [5.24] génère une nouvelle contrainte d'égalité $(u = w\delta_{l,k})$ avec $Var_\mathcal{I}(u) = Var_\mathcal{I}(w\delta_{l,k}) \subseteq \{k\}$, ce qui satisfait l'invariant.
La Règle [5.26] du groupe G_5 génère une contrainte d'égalité $(u = v)$. Supposons que $\exists Z_k \in \mathcal{X}_\mathcal{I}$ et $c_j \in \mathcal{C}_\mathcal{I}$ tels que $k, j \in R$, $Z_k \leqslant_m u$ et $c_j \leqslant_m v$. Néanmoins, selon l'Invariant 6.6.1.1, appliqué à la contrainte $(X_i = u)$, nous avons $i = k$. Or, par hypothèse d'induction, pour la contrainte $(X_i = v)$, i et j ne peuvent pas être tous les deux quantifiés existentiellement. Ainsi, k et j ne peuvent pas être tous les deux quantifiés existentiellement, ce qui satisfait l'invariant. La Règle [5.27] génère une contrainte d'égalité $(u = v)$. Or, u et v proviennent des contraintes $(X = u)$ et $(X = v)$. Ainsi, $\nexists X_i$ tel que $X_i \leqslant_m u$ ou $X_i \leqslant_m v$ par priorité des Règles [5.6] et [5.7]. Ainsi, la Règle [5.27] satisfait l'invariant. Les autres règles du groupe G_5 ne génèrent pas de nouvelles contraintes d'égalité. Ainsi, le groupe G_5 satisfait l'Invariant 6.6.1.3.
Les Règles [5.33] et [5.34] du groupe G_6 ne génèrent pas de contraintes d'égalité non finales. Ainsi, l'invariant est valide. La Règle [5.35] génère une nouvelle contrainte d'égalité $(u_o\delta_o = v)$. Supposons que $\exists Z_k \in \mathcal{X}_\mathcal{I}$ et $c_j \in \mathcal{C}_\mathcal{I}$ tels que, soit $Z_k \leqslant_m u_o\delta_o$ et $c_j \leqslant_m v$, soit $Z_k \leqslant_m v$ et $c_j \leqslant_m u_o\delta_o$. Dans le premier cas, i.e. $Z_k \leqslant_m u_o\delta_o$ et $c_j \leqslant_m v$, selon l'Invariant 6.6.1.1, appliqué à la contrainte maître $(X_o = u_o)$ en tenant compte de δ_o, nous avons $k = m$. Or, par hypothèse d'induction,

pour la contrainte $(X_m = v)$, m et j ne peuvent pas être tous les deux, quantifiés existentiellement. Ainsi, k et j ne peuvent pas être tous les deux quantifiés existentiellement, ce qui satisfait l'invariant. Dans le second cas, i.e. $Z_k \leqslant_m v$ et $c_j \leqslant_m u_o\delta_o$, selon l'Invariant 6.6.1.1, pour la contrainte $(X_m = v)$, nous avons $m = k$. Or, par hypothèse d'induction, pour la contrainte maître $(X_o = u_o)$, en tenant compte de δ_o, m et j ne peuvent pas être tous les deux quantifiés existentiellement, et donc l'invariant reste valable. Nous concluons que dans les deux cas, l'invariant est valide. □

Le quatrième invariant (Invariant 6.6.1.4) déclare que, pour une contrainte de type Sub, si le terme à l'intérieur du Sub contient un indice existentiel i, alors soit cet indice fait partie des indices de l'environnement, soit cet indice est frais mais les indices existentiels des constantes du terme à l'extérieur de Sub sont antérieurs à i. Cet invariant sera utilisé pour les preuves des Propositions 6.6.2.7 et 6.6.2.8 et de l'Invariant 6.6.2.13.

Invariant 6.6.1.4 *Pour une contrainte $t \in Sub(u, E, \mathcal{E}, \mathcal{K})$ ou $t \in Sub_d(u, E, \mathcal{E}, \mathcal{K})$ où $Var^R_{\mathcal{I}}(u) = \{i\}$ nous avons soit i est frais et $(ant(i, j) = j$ si $Var^{\mathcal{C},R}_{\mathcal{I}}(t) = j \in R)$, soit $i \in Var^R_{\mathcal{I}}(\mathcal{E})$.*

Preuve. Nous prouvons que les différentes règles de notre système d'inférence satisfont l'invariant. Les Règles du premier groupe ne génèrent pas de contraintes Sub. Ainsi, l'invariant est valide. La Règle [5.9] génère une contrainte $(t \in Sub(w, E, \mathcal{E}, \mathcal{K}))$ où $w \in E$, et donc $Var_{\mathcal{I}}(w) = \emptyset$. Les autres règles du second groupe ne génèrent pas de contraintes Sub. Ainsi, l'invariant est valide. La Règle [5.15] transforme une contrainte Sub en une contrainte Sub_d tout en conservant le même terme u. L'invariant reste donc valable. Les Règles [5.16], [5.17] et [5.18] transforment une contrainte $(t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$ en une seule contrainte $(t \in Sub(u', E, \mathcal{E}, \mathcal{K}))$ avec $u' \leqslant_m u$. Si $Var^R_{\mathcal{I}}(u') = \{i\}$ alors, $Var^R_{\mathcal{I}}(u) = \{i\}$. Or, par hypothèse d'induction, soit i est frais et $(ant(i, j) = j$ si $Var^{\mathcal{C},R}_{\mathcal{I}}(t) = j \in R)$, soit $i \in Var^R_{\mathcal{I}}(\mathcal{E})$, ce qui satisfait l'invariant. Pour la Règle [5.19], il y a deux cas. Dans le premier, la règle génère une nouvelle contrainte Sub avec une variable d'indice fraîche k, et donc, si $Var^{\mathcal{C},R}_{\mathcal{I}}(t) = j \in R$ alors $ant(i, j) = j$. Ainsi, l'invariant est valide. Dans le second cas, la règle génère une ancienne contrainte qui existe dans $Hist(B)$. Ainsi, par hypothèse d'induction, l'invariant est valide. La Règle [5.20] élimine tout le bloc. Nous concluons que le troisième groupe satisfait l'invariant. Les règles du quatrième groupe ne génèrent pas de contraintes Sub. Seule la Règle [5.25] peut générer une nouvelle contrainte Sub avec une nouvelle variable d'indice quantifiée existentiellement, à l'intérieur du Sub. Dans ce cas, nous avons $(t \in Sub(u, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$ avec $Var^R_{\mathcal{I}}(u) = \{i\}$. Nous devons aussi avoir dans le bloc $(c_i = c_k)$ où $i, k \in R$ afin d'avoir une contrainte de la forme $(t \in Sub(u', E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(u', E, \mathcal{E}, \mathcal{K}))$ avec $Var^R_{\mathcal{I}}(u') = \{k\}$. Or, par hypothèse d'induction, soit i est frais et $(ant(i, j) = j$ si $Var^{\mathcal{C},R}_{\mathcal{I}}(t) = j \in R)$, soit $i \in Var^R_{\mathcal{I}}(\mathcal{E})$. Si i est frais, alors ceci est en contradiction avec $(c_i = c_k)$ dans le bloc. Si $i \in Var^R_{\mathcal{I}}(\mathcal{E})$, alors $ant(i, j) = \{k\}$ si $k \in \mathcal{E}$ et $ant(i, k) = \{i\}$ sinon. Dans les deux cas, nous avons $Var^R_{\mathcal{I}}(u') \in Var^R_{\mathcal{I}}(\mathcal{E})$, ce qui satisfait l'invariant. Seule la Règle [5.31] généralise une contrainte $Sub_d : (t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))$. Or, w provient d'une contrainte sous-maître. Ainsi, $Var^R_{\mathcal{I}}(w) \in Var^R_{\mathcal{I}}(\mathcal{E})$, et l'invariant est donc valide. Seule la Règle [5.33] génère une contrainte Sub_d. Il y a deux cas. Dans le premier, la règle génère une nouvelle contrainte Sub_d avec une variable d'indice fraîche k', et donc, $(ant(i, j) = j$ si $Var^{\mathcal{C},R}_{\mathcal{I}}(t) = j \in R)$. Ainsi, l'invariant est valide. Dans le deuxième cas, la règle génère une ancienne contrainte qui existe dans $Hist(B)$. Par hypothèse d'induction, l'invariant reste donc valide. □

6.6.2 Une borne pour les indices générés par le système d'inférence

Le but de cette section est de borner l'ensemble des indices susceptibles d'être générés par notre système d'inférence. Pour ceci, nous considérons une dérivation, i.e. des applications successives de règles pour notre système de contraintes. Dans cette dérivation, le nombre d'applications des règles d'étiquetage et de modification de contraintes sous-maîtres, i.e. les Règles [5.4], [5.5] et [5.8], est borné. Nous considérons donc une dérivation où il n'existe pas de modification ou d'étiquetage des contraintes sous-maîtres. L'environnement $\mathcal{E}$ est donc fixé pour toutes les règles. Les indices dans $\mathcal{E}$ sont les indices les plus anciennes. Le but de cette section revient donc à borner l'ensemble des indices susceptibles d'être générés par notre système d'inférence tout au long cette dérivation. Nous énonçons donc le théorème principal de cette section :

Théorème 6.6.2.1 *L'ensemble des indices générés par notre système d'inférence est borné.*

Preuve. Pour le prouver, nous avons besoin de quelques définitions pour les variables d'indices. Nous commençons tout d'abord par définir quelques ensembles de ces variables d'indices pour distinguer celles qui sont quantifiés universellement de celles quantifiées existentiellement ou bien celles pour les constantes de celles pour les variables.

Définition 6.6.2.2 *Variables d'indices.*
$Var_{\mathcal{I}}(t)$ *: l'ensemble d'indices dans* t

$Var^{Q}_{\mathcal{I}}(t)$ *: l'ensemble d'indices quantifiés universellement dans* t

$Var^{R}_{\mathcal{I}}(t)$ *: l'ensemble d'indices quantifiés existentiellement dans* t

$Var^{\mathcal{X},R}_{\mathcal{I}}(t) = \bigcup_{X_i \leqslant_m t} Var^{R}_{\mathcal{I}}(X_i)$ *où* $X_i \in \mathcal{X}_{\mathcal{I}}$

$Var^{\mathcal{C},R}_{\mathcal{I}}(t) = \bigcup_{c_i \leqslant_m t} Var^{R}_{\mathcal{I}}(c_i)$ *où* $c_i \in \mathcal{C}_{\mathcal{I}}$

Ensuite, nous définissons d'autres ensembles se basant sur les indices (Définition 6.6.2.3). Ces ensembles seront utilisés après dans la preuve du Théorème 6.6.2.1 en essayant de les borner d'une manière incrémentale.

Définition 6.6.2.3 *Quelques ensembles d'indices.*
Soient :
$V_1 = Var^{\mathcal{C},R}_{\mathcal{I}}(\{t \,|\, t \in Forge(E,\mathcal{K}) \vee t \in Forge_c(E,\mathcal{K}) \vee t \in Sub(u,E,\mathcal{E},\mathcal{K}) \vee t \in Sub_d(u,E,\mathcal{E},\mathcal{K}) \in B\})$

$V_2 = Var^{R}_{\mathcal{I}}(\{u | t \in Sub(u,E,\mathcal{E},\mathcal{K}) \vee t \in Sub_d(u,E,\mathcal{E},\mathcal{K}) \in B \text{ et } \exists X_i \text{ t.q. } X_i \leqslant_m t\})$

$V_3 = Var^{\mathcal{C},R}_{\mathcal{I}}(\{v | u = v \vee v = u \in B \text{ et } \exists X \text{ t.q. } X \leqslant_m u \text{ et } X \in \mathcal{X} \cup \mathcal{X}_{\mathcal{I}}\})$

$V_4 = Var^{R}_{\mathcal{I}}(\mathcal{E})$

Finalement, $V = V_1 \cup V_2 \cup V_3 \cup V_4$.

Nous pouvons maintenant entamer la preuve du Théorème 6.6.2.1. Cette preuve est structurée comme suit :

Nous bornons tout d'abord l'ensemble d'indices quantifiés universellement, et susceptibles d'être générés par notre système d'inférence. Ce résultat est énoncé en Proposition 6.6.2.6.

Ensuite, nous bornons l'ensemble des indices quantifiés existentiellement susceptibles d'être générés par notre système d'inférence. En considérant une contrainte $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$, en fixant t (par la condition $X_i \leqslant_m t$), nous bornons l'ensemble d'indices quantifiés existentiellement qui sont susceptibles d'être générés à l'intérieur du Sub (dans v) par notre système d'inférence. Pour cela, nous bornons d'abord l'ensemble d'indices quantifiés existentiellement indexant des variables à l'intérieur des Sub. Ce résultat est donné par la Proposition 6.6.2.7. Une fois que cet ensemble est borné, nous bornons l'ensemble d'indices quantifiés existentiellement indexant des constantes à l'intérieur des Sub. Ce résultat est donné par la Proposition 6.6.2.8 permettant ainsi de borner l'ensemble d'indices quantifiés existentiellement à l'intérieur des Sub.

Par la suite, nous bornons l'ensemble des indices quantifiés existentiellement dans V défini en Définition 6.6.2.3. Ce résultat est donné par la Proposition 6.6.2.9.

Exemple 6.6.2.4 *Intérêt des clefs autonomes pour la terminaison.*
Nous revenons à la restriction de clefs autonomes introduites dans la Définition 5.6.2.6 pour voir leur intérêt pour la terminaison. Intuitivement, cette restriction exprime le fait qu'il est autorisé que les termes en position clefs peuvent être composés ou bien des listes entières (sous-forme de mpair). Cependant, ce qui est interdit c'est d'avoir des éléments de listes en cette position. Ceci permet de borner le nombre d'indices pouvant être générés à partir des termes en position clefs. En effet, avec cette restriction, les seules indices pouvant être générés à partir des clefs sont issus des ouvertures des mpairs que nous pouvons borner. Par contre, avoir un élément de liste en position clef signifie la présence d'une variable d'indice dont on a du mal à discerner l'origine par rapport aux autres ensembles d'indices que nous voulons borner.

En outre, sans la restriction des clefs autonomes, nous n'avons plus de borne sur l'ensemble de clefs $\mathcal{K}$ puisque cet ensemble pouvait contenir différentes variantes d'une même clef différenciée juste par l'indice, i.e. des clefs de la forme $c_i, .., c_j$ pour des indices i et j. Cependant, avec la restriction des clefs autonomes, l'ensemble $\mathcal{K}$ ne peut pas dépasser en cardinalité l'ensemble de sous-termes du protocole.

Techniquement, supposons que, dans un bloc B, nous avons la contrainte $t \in Sub_d(\{u\}^s_{c_i}, E, \mathcal{E}, \mathcal{K})$ avec i frais et quantifié existentiellement. Cette contrainte a été obtenue par application de la Règle [5.19] à la contrainte $t \in Sub_d(mpair(i, \{u\}^s_{c_i}), E, \mathcal{E}, \mathcal{K})$. Nous supposons aussi que t ne contient aucune variable. La variable d'indice i n'appartient donc pas à V_2 de la Définition 6.6.2.3 et non plus à V de la même définition. En appliquant la Règle [5.17], nous obtenons, dans un des blocs résultants de B, un bloc B' qui contiendrait la contrainte $c_i \in Forge(E, \mathcal{K} \cup \{c_i\})$. Néanmoins, d'après cette contrainte, l'indice i appartient à V_1. Ainsi, V n'est plus stable, il est passé de V à $V \cup \{i\}$. Par conséquent, l'Invariant 6.6.2.15, utilisé pour la preuve de la Proposition 6.6.2.9, n'est plus satisfait.

Exemple 6.6.2.5 *Intérêt de la notion d'antériorité de variables d'indices.*

La notion d'antériorité de variables d'indices a été utilisée essentiellement dans le cas du remplacement d'une variable d'indice par une autre variable d'indice (dans la Règle [5.25]) pour avantager la variable d'indice qui est apparue la première. Supposons que cet avantage n'existe pas et regardons les effets sur notre système de contraintes et par la suite sur nos preuves de terminaison.

Supposons maintenant que dans un certain bloc B de notre système de contraintes, nous avons la contrainte $mpair(i, \langle X, Y_i \rangle) \in Forge(E, \mathcal{K})$ avec $mpair(i, \langle a_i, c_i \rangle) \in E$. Nous nous intéressons à un bloc B' résultant de ce bloc B qui contiendrait les contraintes $X = c_k$ et

$Y_i = a_k$ avec comme préfixe $\forall i\, \exists k$. Ces contraintes sont le résultat de la normalisation de la contrainte initiale.

Nous supposons maintenant que l'étape suivante ajoute la contrainte $t \in Forge(E, \mathcal{K})$ à tous les blocs et donc à B' avec E comportant les deux termes $\{u\}_X$ et $mpair(i, c_i)$. Comme résultat de la normalisation de cette contrainte, nous aurons dans un des blocs résultants de B', les contraintes $t \in Sub(u, E, \mathcal{E}, \mathcal{K}) \wedge X \in Forge(E, \mathcal{K})$ qui résultent du choix de $\{u\}_X$. La deuxième contrainte $X \in Forge(E, \mathcal{K})$ peut donner en choisissant de décomposer $mpair(i, c_i)$, la contrainte $X = c_l$ avec l est quantifiée existentiellement, qui donnera par la suite $X = c_l$. Nous notons ici que cet indice l à ce stade n'appartient pas à l'ensemble d'indices V.

À ce stade, nous avons dans le même bloc les contraintes $X = c_k \wedge Y_i = c_k \wedge c_l = C_k$ et parmi les quantificateurs du préfixe du système $\forall i\, \exists k\, \exists l$. la dernière contrainte est le résultat de l'entrelacement entre la contrainte $X = c_l$ et la contrainte $X = c_k$. Supposons maintenant que nous allons remplacer k par l sans tenir compte de l'antériorité de k. Alors, nous aurons dans le bloc résultant la contrainte $Y_i = c_k$ qui se transforme en $Y_i = c_l$. Par conséquent, l'ensemble d'indice V va être modifié pour lui ajouter l'indice l, ce qui rend l'Invariant 6.6.2.15, utilisé pour la preuve de la Proposition 6.6.2.9, non satisfait.

Nous pouvons ensuite borner l'ensemble d'indices quantifiés existentiellement qui existent dans u pour des contraintes de la forme $(t \in Sub(u, E, \mathcal{E}, \mathcal{K}))$ sans aucune restriction pour le terme t. Ce résultat est énoncé en Proposition 6.6.2.10.

Une fois que l'ensemble d'indices quantifiés existentiellement pour les contraintes $Forge$ et Sub est borné, nous bornons l'ensemble d'indices quantifiés existentiellement pour les contraintes d'égalité. Ce résultat est donné par la Proposition 6.6.2.11 et la Proposition 6.6.2.12.

En conclusion, la preuve du Théorème 6.6.2.1 découle des propositions suivantes :

Proposition 6.6.2.6
L'ensemble d'indices quantifiés universellement, susceptibles d'être générés par notre système d'inférence, est borné.

Proposition 6.6.2.7
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{v \mid (t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ *ou* $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K})) \in B$ *et* $\exists X_i \leqslant_m t\}) \cup Var_{\mathcal{I}}^{R}(\mathcal{E})$ *est borné.*

Proposition 6.6.2.8
$Var_{\mathcal{I}}^{\mathcal{C},R}(\{v \mid (t \in Sub(v, E, \mathcal{E}, \mathcal{K})) \vee (t \in Sub_d(v, E, \mathcal{E}, \mathcal{K})) \in B$ *et* $\exists X_i \leqslant_m t)\}$ *admet une borne.*

Proposition 6.6.2.9
L'ensemble d'indices dans V (défini par la Définition 6.6.2.3) est borné.

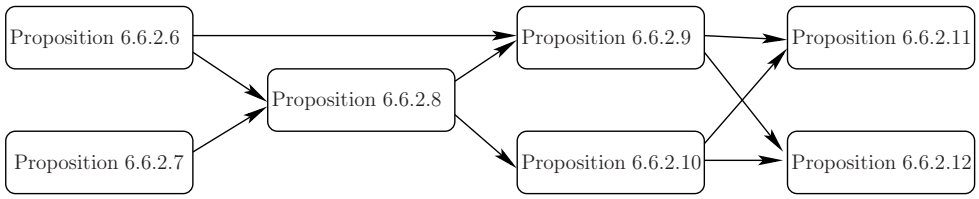
Proposition 6.6.2.10
$Var_{\mathcal{I}}^{R}(\{u \mid t \in Sub(u, E, \mathcal{E}, \mathcal{K}) \vee t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}) \in B\})$ *est borné.*

Proposition 6.6.2.11
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{u \mid (v = u) \vee (u = v) \in B\})$ *est borné.*

Proposition 6.6.2.12
$Var_{\mathcal{I}}^{\mathcal{C},R}(\{u \mid (v = u) \vee (u = v) \in B\})$ *est borné.*

La dépendance entre ces différentes propositions est illustrée par la Figure 6.6.2.

□



Avant de prouver ces différentes propositions, nous allons tout d'abord prouver l'Invariant 6.6.2.13. Cet invariant nous garantira que, pour une contrainte d'égalité susceptible de donner lieu à un remplacement d'indice due à une égalité entre deux constantes indicées par des indices quantifiés existentiellement, l'indice choisi pour le remplacement existe déjà dans V. Par conséquent, V est stable par remplacement d'indices. L'Invariant 6.6.2.13 sera utilisé pour la preuve de l'Invariant 6.6.2.15, et des Propositions 6.6.2.11 et 6.6.2.12.

Invariant 6.6.2.13 *Pour une contrainte non finale ou endormie $(u = v)$ où $Var_{\mathcal{I}}^{\mathcal{C},R}(u) = \{i\}$ et $Var_{\mathcal{I}}^{\mathcal{C},R}(v) = \{j\}$, nous avons $ant(i,j) \in V$ où V est défini en Définition 6.6.2.3.*

Preuve. Les règles du premier et du deuxième groupes ne génèrent pas de contraintes de la forme des contraintes définies dans l'Invariant 6.6.2.13. L'invariant reste donc valide.

Seule la Règle [5.15] génère une contrainte d'égalité $(u = v)$ à partir d'une contrainte $(u \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ où $Var_{\mathcal{I}}^{\mathcal{C},R}(u) = \{i\}$ et $Var_{\mathcal{I}}^{\mathcal{C},R}(v) = \{j\}$. Or, selon l'Invariant 6.6.1.4, soit j est frais et $ant(i,j) = i$, soit $j \in Var_{\mathcal{I}}^{R}(\mathcal{E})$. Nous distinguons donc ces deux cas. Dans le premier, j est frais et $ant(i,j) = i$. Or, $i \in V$ puisque $i \in V_1$. Ainsi, $ant(i,j) \in V$ et l'invariant est valide. Dans le deuxième cas, $j \in Var_{\mathcal{I}}^{R}(\mathcal{E})$. Ainsi, $ant(i,j) = i$ si $i \in Var_{\mathcal{I}}^{R}(\mathcal{E})$ et $ant(i,j) = j$ sinon. Dans les deux cas, $ant(i,j) \in V$ et donc l'invariant reste valable. Le même raisonnement est applicable quand la contrainte $(v \in Sub(u, E, \mathcal{E}, \mathcal{K}))$ est transformée en une contrainte $(u = v)$.

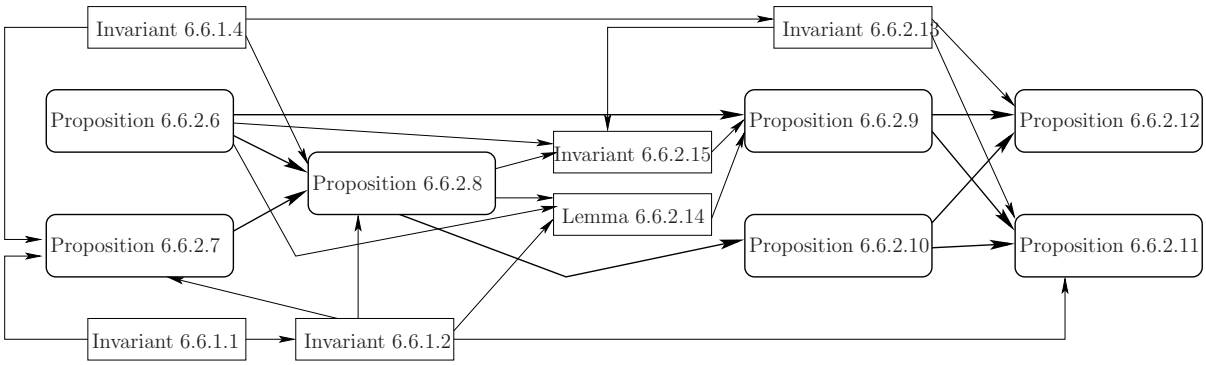
La Règle [5.21] élimine une contrainte. La Règle [5.22] élimine tout le bloc. La Règle [5.23] décompose une égalité entre deux termes en une égalité entre deux sous-termes tout en conservant les mêmes indices. Ainsi, l'invariant reste valide. La Règle [5.24] génère une contrainte d'égalité $(u = v)$ où $Var_{\mathcal{I}}(u) = Var_{\mathcal{I}}(v)$. Ainsi, l'invariant reste valable. La Règle [5.25] remplace un indice par un autre dans tout le bloc. Ceci peut générer une nouvelle contrainte $(u = v')$ à partir d'une contrainte $(u = v)$ où $Var_{\mathcal{I}}(v) = \{j\}$, $Var_{\mathcal{I}}(v') = \{k\}$, $Var_{\mathcal{I}}(u) = \{i\}$ et $i, j, k \in R$. Le bloc doit contenir alors une contrainte $(c_j = c_k)$. Ainsi, $ant(j,k) = k$. Nous distinguons deux cas concernant la contrainte $(u = v')$. Dans le premier cas, $ant(i,k) = k$. Or, par hypothèse d'induction pour la contrainte $(c_j = c_k)$ et puisque $ant(j,k) = k$, alors $k \in V$, ce qui satisfait l'invariant. Dans le deuxième cas, $ant(i,k) = i$. Or, $ant(j,k) = k$. Ainsi, $ant(i,j) = i$ et par la suite, par hypothèse d'induction pour la contrainte $(u = v)$, $i \in V$, ce qui satisfait l'invariant. Ainsi, dans les deux cas, l'invariant est valide.

Seules les Règles [5.26] et [5.27] du cinquième groupe peuvent générer de nouvelles contraintes de la forme $(u = v)$. Or, u et v appartiennent à des contraintes de la forme $(X_i = u)$ ou $(X = u)$ dont les indices quantifiés existentiellement sont déjà comptés dans V. Ainsi, les règles du groupe G_5 valident l'invariant.

Seule la Règle [5.35] peut générer une nouvelle contrainte d'égalité de la forme $(u_o\delta_o = v)$. Supposons que $Var_{\mathcal{I}}^{\mathcal{C},R}(u_o\delta_o) = \{i\}$ et $Var_{\mathcal{I}}^{\mathcal{C},R}(v) = \{j\}$. Nous distinguons deux cas. Dans le premier, cette règle génère une nouvelle contrainte $(u_o\delta_o = v)$ avec une variable d'indice fraîche i et donc $ant(i,j) = j$. Or, $j \in V_3$. Ainsi, $ant(i,j) \in V$, ce qui satisfait l'invariant. Dans le

deuxième cas, la règle génère une ancienne contrainte qui existe déjà dans $Hist(B)$. Ainsi, par hypothèse d'induction, l'invariant reste valide. □

La Figure 6.6.2 détaille la dépendance entre les différentes propositions et les invariants prouvant le Théorème 6.6.2.1.



Nous commençons par la première Proposition 6.6.2.6 qui a pour but de borner l'ensemble d'indices quantifiés universellement. Cette proposition sera utilisée pour la preuve du Théorème 6.6.2.1, de la Proposition 6.6.2.8 et du Lemme 6.6.2.14.

Proposition 6.6.2.6.
L'ensemble d'indices quantifiés universellement, susceptibles d'être générés par notre système d'inférence, est borné.

Preuve. Il n'y a que deux règles qui peuvent générer des indices quantifiés universellement dans les contraintes, i.e. les Règles [5.13] et [5.24]. Ainsi, le nombre d'indices quantifiés universellement est borné par le nombre de contraintes des formes $(mpair(i,t) \in Forge_c(E,\mathcal{K}))$ et $(mpair(j,u) = mpair(k,v))$.

Le nombre de contraintes de la forme $(mpair(i,t) \in Forge_c(E,\mathcal{K}))$ est borné par (le nombre de $mpair$s $*$ le nombre de différents E $*$ le nombre de différents $\mathcal{K}$). Soit st le nombre de sous-termes du protocole. Le nombre de $mpair$s est borné par st. Le nombre de différents E est borné par 2^{st} vu que les termes de E ne contiennent pas d'indices en dehors des $mpair$s selon la restriction de protocoles bien-tagués (voir Définition 5.6.2.3). Le nombre de différents $\mathcal{K}$ est borné par 2^{keys} qui est borné par 2^{st} vu que les clefs ne contiennent pas d'indices en dehors des $mpair$s selon la restriction de clefs autonomes (voir Définition 5.6.2.6).

Le nombre de contraintes de la forme $(mpair(j,u) = mpair(k,v))$ est borné par (le nombre de $mpair$s $*$ le nombre de $mpair$s) qui est borné par $(st * st)$. Nous concluons que l'ensemble d'indices quantifié universellement est borné. □

Ensuite, nous bornons l'ensemble d'indices quantifiés existentiellement. Pour cela, nous commençons en Proposition 6.6.2.7 par borner l'ensemble d'indices quantifiés existentiellement indiçant des variables à l'intérieur de Sub et ceci en fixant le terme t à l'extérieur du Sub. Cette proposition sera utilisée pour la preuve de la Proposition 6.6.2.8 et du Théorème 6.6.2.1.

Proposition 6.6.2.7.
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{v \text{ t.q. } (t \in Sub(v,E,\mathcal{E},\mathcal{K})) \text{ ou } (t \in Sub_d(v,E,\mathcal{E},\mathcal{K})) \in B \text{ et } \exists X_i \leqslant_m t\}) \cup Var_{\mathcal{I}}^{R}(\mathcal{E})$ est borné.

PREUVE. La Règle [5.19] génère un ensemble borné d'indices. En effet, soit st le nombre de sous-termes du protocole. Ainsi, le nombre d'indices quantifiés existentiellement dans v générés à partir des contraintes $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K}))$ est borné par le nombre de contraintes de cette forme, qui est borné par : (le nombre de termes t possibles $*$ le nombre de différents E $*$ le nombre de différents $\mathcal{K}$) $*$ le nombre d'indices quantifiés existentiellement dans v tout en fixant les autres paramètres. Or, $\exists X_i \leqslant_m t$ et selon l'Invariant 6.6.1.2, $i \in Q$. Le nombre d'indices dans t est donc borné. Soit b_1 cette borne. Ainsi, le nombre de termes possibles t est borné par $2^{b_1 * st}$. En outre, le nombre de différents E est borné par 2^{st} vu que les termes dans E n'ont pas d'indices en dehors des $mpair$s selon la restriction de protocoles bien tagués (voir Définition 5.6.2.3). Ensuite, le nombre de différents $\mathcal{K}$ est borné par 2^{keys} qui est borné par 2^{st} puisque les clefs n'ont pas d'indices en dehors de $mpair$s selon la restriction des clefs autonomes (voir Définition 5.6.2.6). Ainsi, en fixant t, E, $\mathcal{K}$ et $\mathcal{E}$ (qui est fixé par hypothèse sur la dérivation choisie), le nombre de nouveaux indices quantifiés existentiellement dans v générés par la Règle [5.19] est le nombre de $mpair$s qui est borné par st.

Nous prouvons maintenant la Proposition 6.6.2.7 en prouvant l'invariant suivant :
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{v \text{ t.q. } (t \in Sub(v, E, \mathcal{E}, \mathcal{K})) \text{ ou } (t \in Sub_d(v, E, \mathcal{E}, \mathcal{K})) \in B \text{ et } \exists X_i \leqslant_m t\}) \cup Var_{\mathcal{I}}^{R}(\mathcal{E})$ est stable ou décroît par notre système de règles d'inférence à l'exception de la Règle [5.19].
Nous désignons par V l'ensemble d'indices définis dans la Proposition 6.6.2.7. Nous rappelons que, dans la dérivation que nous considérons, $\mathcal{E}$ est fixé. Ainsi, $Var_{\mathcal{I}}^{R}(\mathcal{E})$ est fixé. Les Règles du groupe G_1 ne génèrent pas de contraintes de type Sub. Ainsi, l'invariant reste valide.
La Règle [5.9] du groupe G_2 génère une contrainte $(t \in Sub(w, E, \mathcal{E}, \mathcal{K}))$ avec $w \in E$. Ainsi, $Var_{\mathcal{I}}(w) = \emptyset$, ce qui satisfait l'invariant. Les autres règles du groupe G_2 ne génèrent pas de contraintes de type Sub. Nous concluons que l'application des règles du groupe G_2 valident l'invariant.
La Règle [5.15] transforme une contrainte Sub, soit en une autre contrainte de type Sub_d tout en conservant le même u et donc V reste stable, soit en une contrainte d'égalité et donc V peut décroître. Les Règles [5.16], [5.17] et [5.18] transforment une contrainte de type Sub_d en une autre contrainte de type Sub tout en décomposant le terme à l'intérieur du Sub sans générer de nouveaux indices. Ainsi, V reste stable ou décroît. La Règle [5.20] élimine tout le bloc. V reste donc stable ou décroît.
Les Règles du groupe G_4 ne génèrent pas de nouvelles contraintes de type Sub. En outre, la Règle [5.25] ne remplace pas une variable d'indice quantifiée universellement par une autre quantifiée existentiellement. Le seul cas où la Règle [5.25] peut générer une nouvelle variable d'indice à l'intérieur un Sub est de transformer une contrainte $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K}))$ avec $Var_{\mathcal{I}}(v) = \{i\}$ et $i \in R$ en une autre contrainte $(t \in Sub(v', E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v', E, \mathcal{E}, \mathcal{K}))$ avec $Var_{\mathcal{I}}(v') = \{j\}$ et $j \in R$ par application à la contrainte $(c_i = c_j)$. Or, selon l'Invariant 6.6.1.4, i est fraîche ou $i \in Var_{\mathcal{I}}^{R}(\mathcal{E})$. Le premier cas ($i$ est fraîche) est contradictoire avec l'existence de $(c_i = c_j)$. Dans le second cas $(i \in Var_{\mathcal{I}}^{R}(\mathcal{E}))$, la seule manière pour remplacer i par j est d'avoir $ant(i, j) = \{j\}$. Vu que $i \in Var_{\mathcal{I}}^{R}(\mathcal{E})$, ce cas n'est possible que si $j \in Var_{\mathcal{I}}^{R}(\mathcal{E})$. Ainsi, V reste stable ou décroît.
La Règle [5.31] génère une contrainte $(t \in Sub_d(w, E, \mathcal{E}, \mathcal{K}))$. Supposons que $\exists X_i$ t.q. $X_i \leqslant_m w$ et $i \in R$. Or, ceci est contradictoire avec le fait que X_i provient de la contrainte $(X = w)$ par priorité des Règles [5.6] et [5.7]. Les autres règles du groupe G_5 ne génèrent pas de contraintes de type Sub. Ainsi, V reste stable.
La Règle [5.33] génère une contrainte $(t \in Sub_d(u\delta, E, \mathcal{E}, \mathcal{K}))$. Supposons que $\exists Y_j$ t.q. $Y_j \leqslant_m u\delta$ et $j \in R$. Or, selon l'Invariant 6.6.1.1 pour la contrainte maître en tenant compte de δ, nous

avons $j = m$ et donc V reste stable. Les autres règles du groupe G_6 ne génèrent pas de nouvelles contraintes de type Sub. Ainsi, V reste stable. □

Nous bornons ensuite en Proposition 6.6.2.8 l'ensemble d'indices (indiçant les variables et les constantes) quantifiés existentiellement à l'intérieur de Sub et ceci en fixant le terme t à l'extérieur du Sub (avec la condition $X_i \leqslant_m t$). Cette proposition sera utilisée pour la preuve du Théorème 6.6.2.1, du Lemme 6.6.2.14, de l'Invariant 6.6.2.15 et de la Proposition 6.6.2.10.

Proposition 6.6.2.8.
$Var_{\mathcal{I}}^{\mathcal{C},R}(\{v \mid (t \in Sub(v, E, \mathcal{E}, \mathcal{K})) \vee (t \in Sub_d(v, E, \mathcal{E}, \mathcal{K})) \in B \text{ et } \exists X_i \leqslant_m t)\}$ admet une borne.

Preuve. Nous rappelons que pour la dérivation considérée, $\mathcal{E}$ est fixée. Soit b_2 le nombre de contraintes de $\mathcal{E}$ et $id = Var_{\mathcal{I}}^{R}(\mathcal{E})$. Soit st le nombre de sous-termes du protocole.

Le nombre d'indices quantifiés existentiellement dans v à partir des contraintes $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K}))$ est borné par le nombre de contraintes de cette forme, qui est borné par : (le nombre de termes t possibles $*$ le nombre de différents E $*$ le nombre de différents $\mathcal{K}$) $*$ le nombre d'indices quantifiés existentiellement dans v en fixant les autres paramètres. Or, $\exists X_i \leqslant_m t$ et selon l'Invariant 6.6.1.2, $i \in Q$. Le nombre d'indices dans t est alors borné. Soit b_1 cette borne. Ainsi, le nombre de termes possibles t est borné par $2^{b_1 * st}$. En outre, le nombre de différents E est borné par 2^{st} vu que les termes dans E n'ont pas d'indices en dehors des $mpair$s selon la restriction de protocoles bien tagués (voir Définition 5.6.2.3). Ensuite, le nombre de différents $\mathcal{K}$ est borné par 2^{keys} qui est borné par 2^{st} vu que les clefs n'ont pas d'indices en dehors des $mpair$s selon la restriction des clefs autonomes (voir Définition 5.6.2.6).

En fixant t, E, $\mathcal{E}$ et $\mathcal{K}$ pour la contrainte $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K}))$, il y a quatre règles possibles pour générer des indices quantifiés existentiellement dans v, i.e. les Règles [5.19], [5.31], [5.33] et [5.25]. Nous distinguons donc ces quatre cas.

Dans le premier cas, la Règle [5.19] est celle qui génère un nouvel indice quantifié existentiellement. Le nombre d'indices quantifiés existentiellement possibles dans v est alors le nombre de $mpair$s qui est borné par st.

Dans le deuxième cas, la Règle [5.31] est celle qui génère un nouvel indice quantifié existentiellement. Le nombre d'indices quantifiés existentiellement possibles dans v est donc le nombre de contraintes de la forme $(X = v)^{sm}$ qui est borné par b_2.

Dans le troisième cas, la Règle [5.33] est celle qui génère un nouvel indice quantifié existentiellement. Puisque nous générons un seul indice quantifié existentiellement par variable indicée et par contrainte maître pour cette variable, le nombre d'indices quantifiés existentiellement possibles dans v est le nombre de remplacements possibles qui est borné par (le nombre de variables indicées $*$ le nombre de contraintes maîtres). Le nombre de contraintes maîtres est borné par b_2. Le nombre de variables indicées est borné par (le nombre de vecteurs $*$ le nombre d'indices à l'intérieur du Sub pour les variables indicées). Le nombre de vecteurs est fixé à partir de la spécification du protocole. Le nombre d'indices à l'intérieur d'une contrainte Sub pour des variables indicées est la somme du nombre d'indices quantifiés universellement qui est borné selon la Proposition 6.6.2.6 et le nombre d'indices quantifiés existentiellement pour les variables indicées qui est borné selon la Proposition 6.6.2.7. Ainsi, le nombre d'indices quantifiés existentiellement dans v générés par la Règle [5.33] est borné.

Dans le quatrième cas, la Règle [5.25] est celle qui génère un nouvel indice quantifié existentiellement à l'intérieur du Sub. Ceci n'est possible que si cette règle transforme des contraintes $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K}))$ avec $Var_{\mathcal{I}}(v) = \{i\}$ et $i \in R$, en des contraintes $(t \in Sub(v', E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v', E, \mathcal{E}, \mathcal{K}))$ avec $Var_{\mathcal{I}}(v') = \{j\}$ et $j \in R$, par le biais

d'un remplacement par application de la Règle [5.25] à la contrainte $(c_i = c_j)$. Or, selon l'Invariant 6.6.1.4, i est fraîche ou $i \in Var^R_{\mathcal{I}}(\mathcal{E})$. Le premier cas ($i$ est fraîche) est contradictoire avec l'existence de $(c_i = c_j)$. Dans le deuxième cas ($i \in Var^R_{\mathcal{I}}(\mathcal{E})$), la seule manière pour remplacer i par j est d'avoir $ant(i, j) = \{j\}$. Vu que $i \in Var^R_{\mathcal{I}}(\mathcal{E})$, ce cas n'est possible que si $j \in Var^R_{\mathcal{I}}(\mathcal{E})$. Ainsi, le nombre d'indices générés à l'intérieur du Sub dans ce cas est borné par id. □

Nous nous intéressons maintenant, en Proposition 6.6.2.9, à borner l'ensemble d'indices dans V, défini en Définition 6.6.2.3 en montrant que cet ensemble est stable par notre système de règles. Cette proposition sera utilisée pour la preuve des Propositions 6.6.2.11 et 6.6.2.12.

Proposition 6.6.2.9.
L'ensemble d'indices dans V (défini par la Définition 6.6.2.3) est borné.

Preuve. La preuve de la Proposition 6.6.2.9 est une conséquence directe de l'Invariant 6.6.2.15 et du Lemme 6.6.2.14.

Lemme 6.6.2.14 *L'ensemble d'indices dans V (défini dans Définition 6.6.2.3) générés par les règles suivantes :*
La Règle [5.19] pour une contrainte $(t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$ quand $\exists Z_j \leqslant_m t$;

La Règle [5.33] pour une contrainte $(t \in Sub_d(X_m, E, \mathcal{E}, \mathcal{K}))$ quand $\exists Z_j \leqslant_m t$;

La Règle [5.34] ;

La Règle [5.35] pour une contrainte $(X_m = v)$ quand $\exists Z_m \leqslant_m v$

est borné.

Preuve. La Règle [5.19] génère une nouvelle contrainte $t \in Sub(u\delta, E, \mathcal{E}, \mathcal{K})$ qui peut contenir un nouvel indice quantifié existentiellement dans $u\delta$. Or, $\exists Z_j \leqslant_m t$ et grâce à la Proposition 6.6.2.8 et la Proposition 6.6.2.7, l'ensemble d'indices quantifiés existentiellement que peut générer la Règle [5.19] est borné. D'une manière similaire, la Règle [5.33] peut générer de nouveaux indices quantifiés existentiellement à l'intérieur d'une contrainte de type Sub. Cependant, l'ensemble de ces indices est borné selon les Propositions 6.6.2.8 et 6.6.2.7.
La Règle [5.34] peut générer de nouveaux indices quantifiés existentiellement dans une contrainte $u_o\delta_o \in Forge(E', \mathcal{K})$. Or, l'ensemble de ces indices est borné. En effet, le nombre d'indices quantifiés existentiellement dans $u_o\delta_o$ est borné par le nombre de contraintes de la forme $u_o\delta_o \in Forge(E', \mathcal{K})$ qui est borné par : (le nombre de différents K $*$ le nombre de différents E' $*$ le nombre d'indices quantifiés existentiellement dans $u_o\delta_o$ en fixant E' et $\mathcal{K}$). Or, le nombre de différents E est borné par 2^{st} vu que les termes dans E n'ont pas d'indices en dehors des $mpair$s selon la restriction de protocoles bien tagués (voir Définition 5.6.2.3). En outre, le nombre de différents $\mathcal{K}$ est borné par 2^{keys} qui est borné par 2^{st} vu que les clefs n'ont pas d'indices en dehors des $mpair$s selon la restriction des clefs autonomes (voir Définition 5.6.2.6).

Ensuite, le nombre d'indices quantifiés existentiellement dans $u_o\delta_o$ en fixant E' et $\mathcal{K}$ est le nombre de remplacements possibles qui est borné par (le nombre de variables indicées dans les contraintes $Forge$ $*$ le nombre de contraintes maîtres) vu que nous avons exactement un indice frais par variable indicée et par contrainte maître. Le nombre de contraintes maîtres est borné par le nombre de contraintes dans $\mathcal{E}$ qui est fixe. Le nombre de variables indicées dans des contraintes $Forge$ est borné par (le nombre de vecteurs $*$ le nombre d'indices des

variables pour les contraintes de type $Forge$). Or, selon l'Invariant 6.6.1.2, ces indices sont quantifiés universellement. En outre, selon la Proposition 6.6.2.6, l'ensemble des indices quantifiés universellement est borné. Ensuite, le nombre de vecteurs est fixé à partir de spécification du protocole.
Pour la Règle [5.35], en considérant la contrainte $(X_m = v)$ et vu que $\exists Z_m \leqslant_m v$, nous avons $m \in Q$ selon l'Invariant 6.6.1.2. Pour cette règle, les indices quantifiés existentiellement peuvent donc apparaître seulement dans $u_o\delta_o$. Ainsi, le nombre d'indices frais générés par cette règle est borné par le nombre de contraintes de la forme $(u_o\delta_o = v)$ qui est borné par ((st * l'ensemble d'indices quantifiés universellement) * l'ensemble d'indices quantifiés existentiellement dans $u_o\delta_o$ en fixant v). Or, l'ensemble d'indices quantifiés universellement est borné selon la Proposition 6.6.2.6. En outre, l'ensemble d'indices quantifiés existentiellement dans $u_o\delta_o$ en fixant v est borné par (le nombre de variables indicées dans les contraintes d'égalité * le nombre de contraintes maîtres) vu que nous avons exactement un indice frais par variable indicée dans une contrainte d'égalité par contrainte maître. De plus, le nombre de contraintes maîtres est borné par le nombre de contraintes dans $\mathcal{E}$ qui est fixe. D'ailleurs, le nombre de variables indicées dans des contraintes d'égalité de la forme $(X_m = v)$ est borné par (le nombre de vecteurs * le nombre d'indices des variables pour ces contraintes d'égalité). Vu que $m \in Q$ et que le nombre de vecteurs est fixe, selon la Proposition 6.6.2.6, le nombre de variables indicées est borné.
Nous concluons que l'ensemble d'indices dans V générés par ces règles est borné. □

Invariant 6.6.2.15 *V est stable ou diminue par application de notre système d'inférence à l'exception de :*
La Règle [5.19] pour une contrainte $(t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}))$ quand $\exists Z_j \leqslant_m t$;

La Règle [5.33] pour une contrainte $(t \in Sub_d(X_m, E, \mathcal{E}, \mathcal{K}))$ quand $\exists Z_j \leqslant_m t$;

La Règle [5.34] ;

La Règle [5.35] pour une contrainte $(X_m = v)$ quand $\exists Z_m \leqslant_m v$.

Preuve. V_4 est toujours stable vu la dérivation choisie au début de la Section 6.6.2. V_2 est borné selon les Propositions 6.6.2.7 et 6.6.2.8. Ainsi, dans cette preuve, nous considérons cette borne et nous montrons que, pour chaque règle de notre système d'inférence, les indices que nous pouvons générer dans V_1 et V_3 existent toujours dans V_2 et V_4 qui sont fixes, soit dans V_1 et V_3 avant l'application de la règle. Les règles du groupe G_1 ne génèrent pas de nouvelles contraintes de type $Forge$ ou Sub. V_1 reste donc stable. En outre, les règles de ce groupe ne génèrent pas de nouvelles contraintes d'égalité $u = v$ avec $\exists X_i, Y_j \in \mathcal{X}_\mathcal{I}$ t.q. $X_i \leqslant_m u$ et $Y_j \leqslant_m v$. Ainsi, V reste stable ou décroît par le groupe G_1.
Les règles du groupe G_2 ne génèrent pas de contraintes d'égalité. V_3 reste donc stable. La Règle [5.9] transforme une contrainte $Forge$ en une contrainte $Forge$ soit en une contrainte Sub avec le même t dans les deux cas. V_1 reste donc stable. La Règle [5.13] génère une nouvelle contrainte $(t \in Forge(E, \mathcal{K}))$ avec $Var_\mathcal{I}(t) \subseteq \{k\}$ et $k \in Q$. V_1 reste donc stable. La Règle [5.14] élimine tout le bloc. Les autres règles du groupe G_2 transforment une contrainte $(t \in Forge_c(E, \mathcal{K}))$ en une contrainte $(t' \in Forge(E, \mathcal{K}))$ avec $mpair(t', t)$ sans générer de nouveaux indices. V_1 reste donc stable. Ainsi, V est stable par application des règles du groupe G_2.
La Règle [5.15] transforme une contrainte $(t \in Sub(u, E, \mathcal{E}, \mathcal{K}))$ soit en une autre contrainte de type Sub_d avec le même t et donc V_1 reste stable, soit en une contrainte d'égalité $(t = u)$. Supposons que $\exists X_i$ t.q. $X_i \leqslant_m t$ ou $X_i \leqslant_m u$. Nous distinguons deux cas. Dans le premier cas, $X_i \leqslant_m t$. Supposons que $\exists c_j$ t.q. $c_j \leqslant_m u$ et $j \in R$. Or, j appartient toujours à V_2. Ainsi, $V_2 \cup V_3$ reste

stable et donc V est stable. Dans le deuxième cas, $X_i \leqslant_m u$. Supposons que $\exists c_j$ t.q. $c_j \leqslant_m t$ et $j \in R$. Or, j appartient toujours à V_1. Ainsi, $V_1 \cup V_3$ reste stable et donc V est stable. La Règle [5.16] transforme une contrainte de type Sub_d en une autre contrainte de type Sub tout en conservant le même t mais en décomposant le terme à l'intérieur du Sub sans générer de nouveaux indices. V_1 reste donc stable. Il n'y a pas de nouvelles contraintes d'égalité. Ainsi, V_3 reste stable et donc V reste stable. La Règle [5.17] transforme une contrainte de type Sub_d en une contrainte de type Sub tout en conservant le même t et en générant une contrainte de type $Forge$. La dernière contrainte (celle de type $Forge$) utilise la clef b. Selon la Définition 5.6.2.6 de clefs autonomes, $Var_{\mathcal{I}}(b) = \emptyset$. Ainsi, V_1 reste stable et donc V reste stable. Le raisonnement est similaire pour la Règle [5.18]. La Règle [5.19] est appliquée quand $\nexists Z_j \leqslant_m t$ et donc V_2 ne change pas pour cette règle. V_1 reste stable vu que la règle conserve le même t. La Règle [5.20] élimine le bloc. Ainsi, V reste stable par application des règles du groupe G_3.
La Règle [5.21] élimine la contrainte. la Règle [5.22] élimine tout le bloc. V reste donc stable ou décroît. La Règle [5.23] décompose une contrainte d'égalité sans générer de nouveaux indices. V_3 reste donc stable. Cette règle ne génère pas d'autres nouvelles contraintes. V reste donc stable. La Règle [5.24] génère une nouvelle contrainte d'égalité $(u = w\delta_{l,k})$ avec $Var_{\mathcal{I}}(u) = w\delta_{l,k} \subseteq \{k\}$ et $k \in Q$. V_3 reste alors stable et donc V est stable. Ainsi, V reste stable par application des règles du groupe G_4. La Règle [5.25] remplace un indice par un autre, ce qui mène à la génération d'un nouvel indice dans V. Dans ce cas, le bloc doit contenir une contrainte $(c_i = c_j)$ avec $i, j \in R$, $i \in V$, $j \notin V$ et $ant(i, j) = j$. Or, selon l'Invariant 6.6.2.13 pour la contrainte $(c_i = c_j)$, $ant(i, j) \in V$, ce qui contredit le fait que $j \notin V$. Ainsi, V est stable par application de la Règle [5.25].
La Règle [5.26] génère une nouvelle contrainte d'égalité $(u = v)$. Si $\exists c_j$ t.q. $j \in R$ et $c_j \leqslant_m u$ ou $c_j \leqslant_m v$, alors j appartient soit à la contrainte $(X_i = u)$ ou $(X_i = v)$ et donc V_3 ne change pas. Le même raisonnement est valide pour la Règle [5.27]. La Règle [5.29] peut ajouter un indice à V_1. Néanmoins, cet indice provient d'une contrainte sous-maître qui est déjà comptée dans V_4. V reste donc stable. Le même raisonnement est valide pour la Règle [5.31]. La Règle [5.30] ne génère pas de nouvelles contraintes. La Règle [5.32] élimine tout le bloc. Ainsi, V reste stable ou décroît par application des règles du groupe G_5.
La Règle [5.33] est appliquée seulement si $\nexists Z_j$ t.q. $Z_j \leqslant_m t$. V_2 ne change pas. En outre, cette règle génère une nouvelle contrainte de type Sub à partir d'une contrainte Sub_d tout en conservant le même t. V_1 reste donc stable. La Règle [5.35] n'est appliquée que quand $(X_m = v)$ et $\nexists Z_m$ t.q. $Z_m \leqslant_m v$. Supposons donc que cette règle génère une contrainte d'égalité $(u_o\delta_o = v)$ qui peut ajouter un indice à V_3, i.e. $\exists Z_m, c_j$ t.q. $j \in R$ and $Z_m \leqslant_m u_o\delta_o$. j appartient donc déjà à V_3 grâce à la contrainte $(X_m = v)$. Ainsi, V_3 reste stable. De la même manière, l'indice quantifié existentiellement qui peut être généré par ajout de la contrainte $(v \in Forge_c(E'_r, \mathcal{K}_r))$ existe déjà dans V_3 grâce à la contrainte $(X_m = v)$. $V_1 \cup V_3$ reste donc stable. Ainsi, V est stable ou décroît par application des règles du groupe G_6. □

Nous concluons que l'ensemble d'indices de V est borné. □

Nous nous intéressons maintenant, en Proposition 6.6.2.10, à borner l'ensemble d'indices qui existent à l'intérieur du Sub en général, i.e. sans aucune restriction sur le terme à l'extérieur du Sub. Cette proposition sera utilisée pour la preuve du Théorème 6.6.2.1, et des Propositions 6.6.2.11 et 6.6.2.12.

Proposition 6.6.2.10.
$Var^R_{\mathcal{I}}(\{u \mid t \in Sub(u, E, \mathcal{E}, \mathcal{K}) \vee t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}) \in B\})$ est borné.

PREUVE. Considérons une contrainte de la forme $(t \in Sub(v, E, \mathcal{E}, \mathcal{K}))$ ou $(t \in Sub_d(v, E, \mathcal{E}, \mathcal{K}))$. Vu que $Var_{\mathcal{I}}(t)$ est maintenant borné selon la Proposition 6.6.2.9, nous suivons la même preuve que celle de la Proposition 6.6.2.7 afin de prouver que
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{v \text{ t.q. } (t \in Sub(v, E, \mathcal{E}, \mathcal{K})) \text{ ou } (t \in Sub_d(v, E, \mathcal{E}, \mathcal{K})) \in B \text{ et } \exists X_i \leqslant_m t\}) \cup Var_{\mathcal{I}}^{R}(\mathcal{E})$ est borné.
Ensuite, nous suivons la même preuve que celle de la Proposition 6.6.2.8 afin de prouver que
$Var_{\mathcal{I}}^{\mathcal{C},R}(\{v \,|\, (t \in Sub(v, E, \mathcal{E}, \mathcal{K})) \vee (t \in Sub_d(v, E, \mathcal{E}, \mathcal{K})) \in B)\}$ est borné. Nous concluons que
$Var_{\mathcal{I}}^{R}(\{u \,|\, t \in Sub(u, E, \mathcal{E}, \mathcal{K}) \vee t \in Sub_d(u, E, \mathcal{E}, \mathcal{K}) \in B\})$ est borné. □

Nous bornons maintenant en Proposition 6.6.2.11 l'ensemble d'indices indiçant les variables dans des contraintes de type égalité. Cette proposition sera utilisée pour la preuve du Théorème 6.6.2.1 et de la Proposition 6.6.2.12.

Proposition 6.6.2.11.
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{u \,|\, (v = u) \vee (u = v) \in B\})$ est borné.

PREUVE. Nous prouvons la Proposition 6.6.2.12 en prouvant l'invariant suivant :
$Var_{\mathcal{I}}^{\mathcal{X},R}(\{u \,|\, (v = u) \vee (u = v) \in B\})$ est stable ou croît par un ensemble d'indices qui est déjà borné.
Le premier groupe ne génère pas de nouvelles contraintes d'égalité avec de nouveaux indices. Les règles du second groupe ne manipulent pas de contraintes d'égalité. L'invariant est donc valide pour les règles des groupes G_1 et G_2.
La seule règle du troisième groupe qui peut générer une contrainte d'égalité est la Règle [5.15]. Or, les indices possibles quantifiés existentiellement qui peuvent être dans cette contrainte d'égalité appartiennent déjà dans la contrainte *Sub*. Ainsi, l'ensemble des ces indices est borné selon les Proposition 6.6.2.9 et 6.6.2.10 et donc, l'invariant reste valide.
La Règle [5.21] élimine une contrainte. La Règle [5.22] élimine tout le bloc. Si la Règle [5.24] génère un nouvel indice, alors cet indice est quantifié universellement. Ainsi, pour ces règles, l'invariant reste satisfait. La Règle [5.23] décompose une contrainte d'égalité entre deux termes en une autre contrainte d'égalité entre deux sous-termes tout en conservant les mêmes variables d'indices. L'ensemble des indices quantifiés existentiellement reste donc stable et l'invariant est valide. La Règle [5.25] remplace un indice par un autre en utilisant la contrainte $(c_i = c_j)$. Or, selon l'Invariant 6.6.2.13, $ant(i, j) \in V$. Ainsi, même si la Règle [5.25] génère un nouvel indice dans d'autres contraintes d'égalité, alors, cet indice appartient à un ensemble qui est déjà borné vu que cet indice est dans V et V est borné selon la Proposition 6.6.2.9. L'invariant est donc valide.
Seules les Règles [5.26] et [5.27] du cinquième groupe peut générer de nouvelles contraintes d'égalité. Or, les indices possibles dans ces contraintes appartiennent déjà aux contraintes d'égalité dans la partie gauche des règles. L'ensemble des indices est alors stables et l'invariant est donc valide.
Seule la Règle [5.35] du sixième groupe peut générer de nouvelles contraintes d'égalité : $(u_o\delta_o = v)$. Si $\exists Z_j$ t.q. $j \in R$ et $Z_j \leqslant_m v$, alors, selon l'Invariant 6.6.1.2, $j, m \in Q$, ce qui est contradictoire avec $j \in R$. Le même raisonnement est valide pour le cas où $\exists Z_j$ t.q. $j \in R$ et $Z_j \leqslant_m u_o\delta_o$ vu que u_o appartient à la contrainte maître $(X_{i_o} = u_o)$. L'invariant reste donc valide. □

Finalement, nous bornons l'ensemble d'indices (indiçant les variables et les constantes) existants dans les contraintes de type égalité.

Proposition 6.6.2.12.
$Var_{\mathcal{I}}^{\mathcal{C},R}(\{u \,|\, (v = u) \vee (u = v) \in B\})$ est borné.

PREUVE. Nous proposons la Proposition 6.6.2.12 en prouvant l'invariant suivant :
$Var_{\mathcal{I}}^{\mathcal{C},R}(\{u \mid (v = u) \vee (u = v) \in B\})$ est stable ou croît par un ensemble d'indices qui est déjà borné.
Nous nous focalisons seulement sur des contraintes d'égalité $(v = u)$ qui ne contiennent pas de variables indicées. Si $\exists X_i$ t.q. $X_i \leqslant_m u$ ou $X_i \leqslant_m v$ alors $Var_{\mathcal{I}}^{\mathcal{C},R}(\{u \mid (v = u) \vee (u = v) \in B\})$ est borné selon la Proposition 6.6.2.9. Supposons que $(v = u)$ ne contient pas de variables indicées.

Le premier groupe ne génère pas de nouvelles contraintes d'égalité avec de nouveaux indices. Les règles du second groupe ne manipulent pas de contraintes d'égalité. L'invariant est donc valide par application des règles des groupes G_1 et G_2.
La seule règle du troisième groupe qui peut générer une contrainte d'égalité est la Règle [5.15]. Or, les indices quantifiés existentiellement qui peuvent être dans cette contrainte d'égalité appartient déjà dans la contrainte Sub. Ainsi, l'ensemble de ces indices est borné selon les Propositions 6.6.2.9 et 6.6.2.10, et donc l'invariant est valide.
La Règle [5.21] élimine une contrainte. La Règle [5.22] élimine tout le bloc. Si la Règle [5.24] génère un nouvel indice, alors il est quantifié universellement. Ainsi, pour ces règles, l'invariant est valide. La Règle [5.23] décompose une contrainte d'égalité entre deux termes en une autre contrainte d'égalité entre deux sous-termes tout en conservant les mêmes variables d'indices. L'ensemble d'indices quantifiés existentiellement reste donc stable et l'invariant est alors valide.
La Règle [5.25] remplace un indice par un autre en utilisant la contrainte $(c_i = c_j)$. Or, selon l'Invariant 6.6.2.13, $ant(i, j) \in V$. Ainsi, même si la Règle [5.25] génère de nouveaux indices dans d'autres contraintes d'égalité, alors, ces indices appartiennent à un ensemble déjà borné vu que ces indices sont dans V et que V est borné selon la Proposition 6.6.2.9. Ainsi, l'invariant est valide.
Seules les Règles [5.26] et [5.27] du cinquième groupe peuvent générer de nouvelles contraintes d'égalité. Or, les indices possibles dans ces contraintes appartiennent déjà aux contraintes d'égalité à gauche de ces règles. L'ensemble des indices est alors stable et donc l'invariant est valide.
Seule la Règle [5.35] du sixième groupe peut générer de nouvelles contraintes d'égalité : $(u_o\delta_o = v)$. Si $\exists c_j$ t.q. $j \in R$ et $c_j \leqslant_m v$, alors j est déjà compté vu qu'il appartient à la contrainte $(X_m = v)$ et que, selon la Proposition 6.6.2.11, l'ensemble des ces indices est borné. Si $\exists c_j$ t.q. $j \in R$ et $c_j \leqslant_m u_o\delta_o$, alors nous avons un indice frais par variable indicée par contrainte maître. Ainsi, le nombre d'indices quantifiés existentiellement générés de cette manière est borné par (le nombre de variables indicées $*$ le nombre de contraintes maîtres). Or, le nombre de contraintes maîtres est borné par le nombre de contraintes dans $\mathcal{E}$ qui est fixe. En outre, le nombre de variables indicées est borné par (le nombre de vecteurs $*$ le nombre d'indices des variables dans des contraintes d'égalité). De plus, le nombre de vecteurs est défini par la spécification du protocole. En outre, le nombre d'indices des variables dans les contraintes d'égalité est la somme des indices quantifiés universellement et ceux quantifiés existentiellement. L'ensemble des indices quantifiés universellement est borné selon la Proposition 6.6.2.6. L'ensemble de ceux quantifiés existentiellement est borné selon la Proposition 6.6.2.11. Nous concluons que l'invariant est satisfait par la Règle [5.35]. □

6.6.3 Terminaison pour les protocoles bien tagués avec clefs autonomes

Nous étendons les définitions données en Section 6.5.1 pour tenir compte des indices et des $mpair$. La définition du rang d'un terme reste inchangée mais elle considère aussi des variables indicées.

Définition 6.6.3.1 *Rang d'un terme.*
Le rang d'un terme t est défini comme suit :
– $r(X) = max\{l \mid X \sqsubset_l Y, Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}\}$ *pour* $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$
– $r(t) = max\{r(Y) \mid Y < t, Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}\}$
Pour une variable $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$*, si* $\nexists Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$*, t.q.* $X \sqsubset Y$*, alors,* $r(X) = 0$.

La définition de la taille de terme est aussi modifiée pour tenir compte de l'opérateur *mpair*.

Définition 6.6.3.2 *Taille d'un terme.*
Nous définissons la taille d'un terme t, notée $|t|$*, comme suit :*
– $|t| = 1$ *pour* $t \in \mathcal{X} \cup \mathcal{C}$
– $|f(u_1, .., u_m)| = 1 + |u_1| + .. + |u_m|$
– $|h(u)| = 2 + |u|$ *pour* $h \in H$
– $|mpair(k, u)| = 2 + |u\delta|$ $\forall \delta$
Nous étendons cette définition aux ensembles de termes : $|E| = \Sigma_{t \in E}|t|$ *pour* $E \subset T$.

La définition du poids d'un terme est aussi modifiée pour tenir compte de l'opérateur *mpair*.

Définition 6.6.3.3 *Poids d'un terme.*
Soit p la taille du protocole, i.e. la somme des tailles des messages. Nous définissons le poids d'un terme t, noté $\|t\|$*, comme suit :*
– $\|X\| = p^{r(X)+1}$ *pour* $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$
– $\|c\| = 1$ *pour* $c \in \mathcal{C} \cup \mathcal{C}_\mathcal{I}$
– $\|f(u_1, .., u_m)\| = 1 + \|u_1\| + .. + \|u_m\|$
– $\|h(u)\| = 2 + \|u\|$ *pour* $h \in H$
– $\|mpair(k, u)\| = 2 + \|u\delta\|$ $\forall \delta$
Nous étendons cette définition aux ensembles de termes : $\|E\| = 1 + \Sigma_{t \in E}\|t\|$ *pour* $E \subset T$.

Nous conservons la même définition du poids d'une contrainte élémentaire que la Définition 6.5.1.4. Avant de définir le poids du bloc de contraintes, nous prouvons tout d'abord la Proposition 6.6.3.4.

Proposition 6.6.3.4 *Le nombre d'application de la Règle [5.26] est borné.*

Preuve. La Règle [5.26] est appliquée pour chaque paire de contraintes de la forme $X_i = u$ et $X_i = v$ (grâce à Hyp[5.26]). Cependant, selon le Théorème 6.6.2.1, l'ensemble d'indices généré par notre système d'inférence est borné. Ainsi, le nombre de paires de contraintes pour la même variable X_i est borné. Par conséquent, le nombre d'applications de la Règle [5.26] est borné. □

Le poids d'un bloc de contraintes peut être alors défini comme suit :

Définition 6.6.3.5 *Poids d'un bloc de contraintes.*
Nous rappelons que, selon le Théorème 6.6.2.1, l'ensemble d'indices générés par notre système d'inférence est borné. Ainsi, soit Na *le nombre maximal d'applications de la Règle [5.26] (Voir Proposition 6.6.3.4, ce nombre est borné). Soit* NcE *le nombre maximal de contraintes d'égalité de la forme* $X = u$ *où* $X \in \mathcal{X}$*,* $NcEF$ *le nombre maximal de contraintes de la forme* $X = u$ *ou* $X \in Forge_c(E, \mathcal{K})$ *où* $X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$*, et* NcN *le nombre maximal de contraintes* négatives *de type* $Forge$ *ou d'égalité.*
Considérons un bloc de contraintes : $B = ctr_1 \wedge ..ctr_l$*. Nous désignons par* Fc_B *le nombre de contraintes finales dans B,* Nc_B *le nombre de contraintes négatives dans B,* Sc_B *le nombre de*

contraintes sous-maîtres dans B, MEc_B le nombre d'égalités maîtres dans B, MFc_B le nombre de contraintes maîtres dans B de type $Forge$, Ec_B le nombre de contraintes d'égalité de la forme $X = u$ ($X \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$) dans B et NR_B le nombre d'applications de la Règle [5.26]. Nous désignons par $\langle\rangle$ l'ordre lexicographique et [] le multi-ensemble. Nous définissons le poids de B comme suit :

$$\|B\| = \langle Na - NR_B,\ NcEF - Fc_B,\ NcN - Nc_B,\ NcE - Sc_B,\ NcEF - MEc_B,\\ NcEF - MFc_B,\ [\|ctr_i\|]_{ctr_i \in B},\ NcE - Ec_B\rangle$$

Pour le poids du système de contraintes, nous gardons la même Définition 6.5.1.6. Nous pouvons maintenant prouver le Lemme 6.2.0.8 en montrant que le poids du système de contraintes diminue pour chaque application de nos règles.

Preuve. Preuve du Lemme 6.2.0.8.

Nous prouvons que chaque règle de notre système d'inférence diminue le poids de tout le système de contraintes S. En considérant une règle r, nous montrons que $\|\text{post}(r)\| < \|\text{pre}(r)\|$. Nous désignons toujours le bloc traité par ces règles par B. Tout d'abord, la Règle [5.26] diminue le poids du bloc auquel elle est appliquée vu que cette règle diminue $Na - NR_B$. Ensuite, les Règles [5.3], [5.14], [5.20], [5.22] et [5.32] éliminent un bloc d'un système de contraintes vu qu'elles mènent toutes à $\bot$. Ainsi, elles font diminuer $\|S\|$. En outre, les Règles [5.21], [5.25] et [5.30] éliminent une contrainte du bloc traité B vu qu'elles mènent à $\top$. Ainsi, $\|B\|$ diminue.

La Règle [5.1] transforme une contrainte d'égalité $ctr \in B$ en une autre contrainte d'égalité de la forme $(X = u)$ où $X \in X \cup \mathcal{X}_\mathcal{I}$. Le seul paramètre qui change pour le poids de B est Ec_B qui augmente et par la suite $NcE - Ec_B$ diminue. Ainsi, $\|B\|$ diminue et ainsi fait $\|S\|$. Pour la Règle [5.2], nous avons $\|X\| > \|u\|$ vu que $X = u \in B$. En effet, $\|u\| \leq |u| * p^{max\{\mathrm{r}(Y)|Y<u, Y \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}\}} \leq p^{\mathrm{r}(X)+1}$. Puis, $\|Y = X\| = \|Y\| + \|X\| > \|Y\| + \|u\| = \|Y = u\|$. Ainsi, $\|Y = X\| > \|Y = u\|$ et par conséquent, $\|B\|$ et $\|S\|$ diminuent. La Règle [5.4] ajoute une nouvelle contrainte maître et change uniquement les contraintes endormies. Puis, le seul paramètre qui change pour le poids de B est $NcEF - MEc_B$ soit $NcEF - MFc_B$. Ainsi, $\|B\|$ et $\|S\|$ diminuent. La Règle [5.5] transforme une contrainte maître de la forme $Forge$ en une contrainte d'égalité maître. Puis, elle fait augmenter $NcEF - MFc_B$ mais fait diminuer $NcEF - MEc_B$. Ainsi, $\|B\|$ et $\|S\|$ diminuent. Pour la Règle [5.6], nous avons $\|u\lambda\| < \|u\|$ où $\lambda = [Y_j \leftarrow Z]$. En effet, vu que $Y_j = Z \in B$, nous avons $\mathrm{r}(Z) < \mathrm{r}(Y_j)$ qui mène à $\|Z\| < \|Y_j\|$. Ainsi, $\|X = u\lambda\| < \|X = u\|$ et par la suite, $\|B\|$ diminue. Pour la Règle [5.7], nous raisonnons d'une manière similaire à la Règle [5.6] pour prouver que $\|X = u\lambda\| < \|X = u\|$. En outre, $\|Y_j = Z\| < \|X = u\|$. En effet, $Y_j < u$ et par conséquent, $\|Y_j\| < \|u\|$. Ensuite, Z est une variable fraîche et vu que $X = u \in B$ alors, $\mathrm{r}(Z) < \mathrm{r}(X)$. Ainsi, $\|Z\| < \|X\|$ et par la suite, $\|Y_j = Z\| < \|X = u\|$. Nous concluons que $\|B\|$ diminue vu que les autres paramètres ne changent pas. La Règle [5.8] ajoute une contrainte sous-maître au bloc B. Ainsi, $NcE - Sc_B$ diminue et par la suite $\|B\|$ diminue.

Pour la Règle [5.9], $\|\text{pre}(R\ [5.9])\| > \|\text{post}(R\ [5.9])\|$.
En effet, $\|\text{pre}(R\ [5.9])\| = \langle k, \|t\| + \|E\| + |E| + 1\rangle$
et $\|\text{post}(R\ [5.9])\| = [\langle k, \|t\| + \|E\| + |E|\rangle, \langle k, \|t\| + \|w\| + |E| + 1\rangle]$, où $k = st - \#\mathcal{K}$ et $w \in E$ et par conséquent, $\|w\| < \|E\|$. En outre, la Règle [5.9] ne change pas les autres paramètres du poids de B. Ainsi, $\|B\|$ diminue.
Pour la Règle [5.10], $\|\text{pre}(R\ [5.10])\| = \langle k, 1 + \|t_1\| + .. + \|t_m\| + \|E\| + |E|\rangle$ où $k = st - \#\mathcal{K}$. En outre, nous avons $\|\text{post}(R\ [5.10])\| = [\langle k, \|t_i\| + \|E\| + |E| + 1\rangle]_{t_i < \langle t_1 .. t_m\rangle}$ et par la suite, $\|\text{pre}(R\ [5.10])\| > \|\text{post}(R\ [5.10])\|$. Ensuite, la Règle [5.10] ne change pas les autres paramètres du poids de B et par conséquent, $\|B\|$ diminue. Nous montrons d'une manière si-

milaire à la Règle [5.10] que, pour la Règle [5.11], $\|B\|$ diminue. Pour la Règle [5.12], nous avons $\|\text{pre}(R\ [5.12])\| > \|\text{post}(R\ [5.12])\|$. En effet, $\|\text{pre}(R\ [5.12])\| = \langle k, 2 + \|t\| + \|E\| + |E|\rangle$ et $\|\text{post}(R\ [5.12])\| = \langle k, 1 + \|t\| + \|E\| + |E|\rangle$, où $k = st - \#\mathcal{K}$. En outre, la Règle [5.12] ne change pas les autres paramètres du poids de B et par conséquent $\|B\|$ diminue. Nous montrons d'une manière similaire à la Règle [5.13] que, pour la Règle [5.12], $\|B\|$ diminue.

Pour la Règle [5.15], nous avons $\|\text{pre}(R\ [5.15])\| = \langle k, 1 + \|t\| + \|u\| + |E|\rangle$ et les deux cas pour post(R [5.15]) : soit $\|\text{post}(R\ [5.15])\| = \langle k, \|t\| + \|u\|\rangle$,
soit $\|\text{post}(R\ [5.15])\| = [\langle k, \|t\| + \|u\|\rangle, \langle k, \|t\| + \|u\| + |E|\rangle]$. Dans les deux cas, $\|\text{pre}(R\ [5.15])\| > \|\text{post}(R\ [5.15])\|$. En outre, vu que cette règle ne change pas les autres paramètres du poids de B, $\|B\|$ diminue. Nous montrons d'une manière similaire à la Règle [5.10] que, pour la Règle [5.16], $\|B\|$ diminue. Pour la Règle [5.17], nous avons $\|\text{pre}(R\ [5.17])\| = \langle k, 1 + \|t\| + \|u\| + \|b\| + |E|\rangle$ où $k = st - \#\mathcal{K}$. Ensuite, $\|\text{post}(R\ [5.17])\| = [\langle k, 1 + \|t\| + \|u\| + |E|\rangle, \langle k', 1 + \|b\| + \|E\| + |E|\rangle]$ où $k' = st - \#\mathcal{K} \cup \{\{u\}_p^p\}$. Vu que $k' < k$, alors $\|\text{pre}(R\ [5.17])\| > \|\text{post}(R\ [5.17])\|$. Nous montrons d'une manière similaire à la Règle [5.17] que, pour la Règle [5.18], $\|S\|$ diminue. Pour la Règle [5.19], nous avons tout d'abord, $\|\text{pre}(R\ [5.19])\| = \langle k, 2 + \|t\| + \|u\delta\| + |E|\rangle$ où $k = st - \#\mathcal{K}$. Ensuite, $\|\text{post}(R\ [5.19])\| = \langle k, 1 + \|t\| + \|u\delta\| + |E|\rangle$. Ainsi, $\|\text{pre}(R\ [5.19])\| > \|\text{post}(R\ [5.19])\|$ et vu que la règle ne change pas les autres paramètres du poids de B, $\|B\|$ diminue.

Pour la Règle [5.23], $\|\text{pre}(R\ [5.23])\| = \langle k, 2 + \|u_1\| + .. + \|u_m\| + \|w_1\| + .. + \|w_m\|\rangle$. Ensuite, $\|\text{post}(R\ [5.23])\| = [\langle k, \|u_i\| + \|w_i\|\rangle]_{i=1..m}$ où $k = st - \#\mathcal{K}$. Ainsi, $\|\text{pre}(R\ [5.23])\| > \|\text{post}(R\ [5.23])\|$ et vu que les autres paramètres du poids de B ne changent pas, alors $\|B\|$ diminue. Pour la Règle [5.24], nous avons $\|\text{pre}(R\ [5.24])\| > \|\text{post}(R\ [5.24])\|$.
En effet, $\|\text{pre}(R\ [5.24])\| = \langle k, 4 + \|u\delta\| + \|w\delta_{l,k}\delta\|\rangle$. Puis, $\|\text{post}(R\ [5.24])\| = \langle k, \|u\delta\| + \|w\delta_{l,k}\delta\|\rangle$. Ainsi, puisque les autres paramètres du poids de B ne changent pas, alors $\|B\|$ diminue.

Nous montrons d'une manière similaire à la Règle [5.2] que, pour la Règle [5.27], $\|B\|$ diminue vu que $\|X\| > \|u\|$ et par conséquent, $\|X = v\| > \|u = v\|$. Nous montrons d'une manière similaire à la Règle [5.27] que, pour la Règle [5.28], $\|B\|$ diminue vu que $X_i = u \in B$ et ainsi $\|X_i\| > \|u\|$ et par la suite $\|X_i \in Forge_c(E', \mathcal{K})\| > \|u \in Forge_c(E', \mathcal{K})\|$. La même preuve est valide pour la Règle [5.29]. Pour la Règle [5.31], vu que $X = w \in B$, $\|X\| > \|w\|$. Ainsi, $\|t \in Sub_d(X, E', \mathcal{E}, \mathcal{K})\| > \|t \in Sub_d(w, E', \mathcal{E}, \mathcal{K})\|$. En outre, cette règle ne change pas les autres paramètres du poids de B et ainsi $\|B\|$ diminue.

La Règle [5.33] ajoute une nouvelle contrainte finale au bloc traité B, ce qui fait augmenter $NcEF - Fc_B$. Ainsi, $\|B\|$ diminue. La Règle [5.34] ajoute au bloc B une contrainte négative soit une contrainte finale. Dans les deux cas, $\|B\|$ diminue. D'une manière similaire, la Règle [5.35] fait diminuer $\|B\|$. □

6.7 Test de satisfaisabilité

Nous nous intéressons dans cette section au Lemme 6.2.0.9, introduit en Section 6.2 :

Lemme 6.2.0.9 Satisfaisabilité de la forme normale.

Quand l'Algorithme 6.2 est appliqué à un protocole P bien-tagué avec clefs autonomes, la satisfaisabilité du système de contraintes normalisé peut être décidée.

Comme résultat de l'Algorithme 6.2 de la Section 6.2, nous avons un système de contraintes normalisé F qui caractérise une exécution du protocole. Ce système de contraintes est de la forme $F = \forall Q \exists R\, B_1 \vee .. \vee B_p$. Les contraintes des différents blocs sont de type $Forge_c$ ou des

égalités, toutes les deux considérant des variables (indicées ou non indicées).

Pour prouver ce lemme, nous montrerons qu'il existe une limite calculable pour la valeur e de n (le paramètre des listes), notée e_{max}, pour laquelle la satisfaisabilité peut être vérifiée. Pour arriver à ce résultat, nous commençons tout d'abord par continuer la normalisation du système de contraintes F par entrelacements avec les différents termes que peut avoir comme valeur une variable indicée. Ces candidats de valeurs d'une certaine variable sont appelés des *patterns* et sont déduits des différents termes du système de contraintes F. L'idée derrière ces patterns est de définir un ensemble contrôlable de valeurs possibles pour chacune des variables indicées. Nous définissons pour cela des règles d'entrelacements semblables à celles du groupe G_6 de notre système d'inférence sauf qu'ici, nous n'utilisons plus uniquement les valeurs de contraintes maîtres mais plutôt les patterns.

Une fois la normalisation de la première étape effectuée, nous obtenons un système de contraintes F' dans lequel nous transformons toutes les contraintes de type $Forge_c$ en des contraintes de type égalité. Ceci est effectué par le choix de termes différents de tous les patterns qui sont susceptibles d'être utilisés dans les contraintes d'égalité. Nous aurons un terme choisi pour chaque vecteur de variable. Le système de contraintes résultant est noté F".

La dernière étape consiste à trouver un entier e_{max} pour le système de contraintes F", pour lequel nous pouvons vérifier la satisfaisabilité. Cet entier dépend essentiellement du nombre de patterns différents, du nombre d'ensembles de connaissances de l'intrus, et le nombre de vecteurs de variables.

6.7.1 Première étape : la normalisation

Nous commençons tout d'abord par la première étape de la preuve de satisfaisabilité, qui est la normalisation du système de contraintes $F = \forall Q \exists R\, B_1 \vee .. \vee B_p$ en utilisant des *patterns*.

Nous allons construire un autre système de contraintes F' dont la satisfaisabilité est plus facile à vérifier. Pour cela, nous choisissons deux variables d'indices q et r qui seront fixées dans toute la suite de la Section 6.7. Ces indices seront utilisés pour les remplacements des indices dans $Q \cup R$. Nous allons définir la notion de pattern par rapport à un terme (Définition 6.7.1.1) et ensuite par rapport à un système de contraintes (Définition 6.7.1.3).

Définition 6.7.1.1 *Pattern d'un terme.*

Un terme u est dit pattern d'un terme v pour une variable d'indice i, noté $u \ll_i v$, ssi ($u \leqslant_m v$ et $Var_{\mathcal{I}}(v) \subset \{i\}$) soit il existe $mpair(k,t) \leqslant_m v$ tel que $u \ll_i t\delta_{k,i}$.

Notons ici que, par convention, l'indice i ne doit pas être égal à m car la notation $\leqslant_m$ est réservée à la notation de sous-terme sans traverser de *mpair*. Nous citons ci-dessous deux exemples de ces patterns.

Exemple 6.7.1.2 *Pattern de termes.*

$X_i \leqslant_i \{mpair(k, X_k)\}_{c_j}$

$c_j \leqslant_j \{mpair(k, X_k)\}_{c_j}$

Nous définissons maintenant les patterns d'un système de contraintes en Définition 6.7.1.3. L'intuition derrière la définition de patterns d'un système de contraintes est de donner l'ensemble des valeurs possibles d'une variable. Cet ensemble de valeurs possibles sera utilisé pour des remplacements des variables indicées à l'intérieur des blocs du système de contraintes F.

Définition 6.7.1.3 *Patterns d'un système de contraintes.*

Étant donné un système de contraintes $F = \forall Q \exists R\, B_1 \vee .. \vee B_p$, *soit* $\bar{\delta_q}$ *le remplacement* $\delta_{Q\cup R,q}$ *et* $\bar{\delta_r}$ *celui* $\delta_{Q\cup R,r}$. *Nous désignons par* $Patt(F)$ *l'ensemble de tous les patterns de* F *défini par :*

$$Patt(F) = \left\{u \,|\, \exists v \in \mathcal{T} \text{ dans } F \text{ t.q. } u \notin \mathcal{X}_{\mathcal{I}},\, Var^{\mathcal{X}}_{\mathcal{I}}(u) \subseteq \{q\} \text{ et, } u \ll_q v\bar{\delta_q} \text{ ou } u \ll_r v\bar{\delta_r}\right\}$$

Les valeurs possibles des variables indicées peuvent être quantifiées universellement ou bien existentiellement. Ensuite, nous avons vu dans un Invariant précédent, plus particulièrement l'Invariant 6.4.1.1, que pour tout terme t, nous avons $\#Var_{\mathcal{I}}(t) \leq 1$. Pour cela, nous considérons donc ces deux variantes de patterns : une quantifiée universellement et une autre quantifiée existentiellement. D'où la condition $u \ll_q v\bar{\delta_q}$ ou $u \ll_r v\bar{\delta_r}$ de la Définition 6.7.1.3.

En outre, puisqu'un pattern est supposé représenter une variable indicée, l'indice des variables existant dans ce pattern est quantifié universellement. D'où la condition $Var^{\mathcal{X}}_{\mathcal{I}}(u) \subseteq \{q\}$ de la Définition 6.7.1.3.

Nous équipons l'ensemble $Patt(F)$ d'un ordre $\lhd$ qui sera fixé dans tout ce qui suit, i.e. dans le reste de la Section 6.7. Cet ordre permet d'avoir une préférence sur l'ensemble des patterns lors d'un remplacement d'une variable. En effet, si nous choisissons un pattern pour un remplacement d'une variable, nous garantissons qu'aucune valeur (pattern) d'ordre inférieur (selon $\lhd$) ne puisse être candidate comme valeur de cette même variable. D'où l'introduction des contraintes négatives pour les Règles [6.1] et [6.2], définies ci-dessous.

En outre, pour tout terme $u \in \mathcal{T}$, nous désignons par $u\bar{\delta}$ le pattern de $Patt(F)$ ayant l'ordre le plus élevé entre les patterns $u\bar{\delta_q}$ et $u\bar{\delta_r}$ selon $\lhd$. Cette valeur maximale par rapport à $\lhd$ est dite le pattern maximal de u. Elle sera utilisée dans le cas où la variable à remplacer admet déjà une contrainte d'égalité qui assigne une valeur à la variable considérée. Dans ce cas, nous utilisons comme ensemble de patterns possibles pour la variable en question ceux qui sont d'ordre inférieur au pattern maximal de la valeur assignée de la variable. D'où la condition $u' \lhd u\bar{\delta}$ dans la définition de la Règle [6.1] dans le cas d'une contrainte d'égalité.

Nous introduisons donc ici nos deux règles d'entrelacements utilisant les patterns. Ces règles assurent que notre système de contraintes utilise toujours les plus petits patterns ayant l'ordre le plus élevé. Les Règles [6.1] et [6.1] sont semblables à celles du groupe G_6 de la Section 5.8.2 du Chapitre 5. Notons que nous ne considérons pas ici l'entrelacement d'une contrainte Sub vu qu'à la fin de la normalisation, notre algorithme nous garantie que notre système de contraintes est normalisé, et qu'il contient donc uniquement des contraintes $Forge_c$ et des contraintes d'égalité.

$$(X_m = u)^{\cdot} \longrightarrow \bigvee_{u' \lhd u\bar{\delta}} \exists k \qquad [6.1]$$
$$((X_m = u'\delta^k_{q,m})^{\cdot} \wedge \bigwedge_{v \lhd u'} \left(\forall k'.X_m \neq v\delta^{k'}_{q,m}\right) \wedge (u'\delta^k_{q,m} = u))$$

$$(X_m \in Forge_c(E,\mathcal{K}))^{\cdot} \longrightarrow \bigvee_{u' \in Patt(F)} \exists k((X_m = u'\delta^k_{q,m})^{\cdot} \qquad [6.2]$$
$$\wedge \bigwedge_{v \lhd u'} \left(\forall k'.X_m \neq v\delta^{k'}_{q,m}\right) \wedge u'\delta^k_{q,m} \in Forge_c(E,\mathcal{K}))$$
$$\vee (X_m \in Forge_c(E,\mathcal{K}))^{\cdot} \wedge \bigwedge_{v \in Patt(F)} \left(\forall k'.X_m \neq v\delta^{k'}_{q,m}\right)$$

La Règle [6.1] considère une contrainte d'égalité. Elle choisit des patterns de $Patt(F)$ qui sont inférieurs au pattern maximal de sa valeur. Pour chacun de ces patterns choisis, la valeur de la variable doit être différente de tous les patterns inférieurs (selon $\lhd$) au pattern choisi. Pour le cas de remplacement pour une contrainte $Forge_c$ (Règle [6.2]), nous utilisons tous les patterns de $Patt(F)$ avec la même condition : une fois que le pattern est choisi, la variable doit être différente de tous les patterns inférieurs à ce pattern (selon $\lhd$).

Comme pour le groupe G_6 des règles d'entrelacements, nous ajoutons des contrôles pour ne pas boucler. Ainsi, la Règle [6.1] ne peut pas être utilisée si le bloc contenait déjà $\bigwedge_{v \lhd u\bar{\delta}} \left(\forall k'.X_m \neq v\delta^{k'}_{q,m}\right)$. De même, la Règle [6.2] ne peut pas être utilisée si le bloc contenait déjà $\bigwedge_{v \in Patt(F)} \left(\forall k'.X_m \neq v\delta^{k'}_{q,m}\right)$. En outre, ces deux règles mettent à jour implicitement l'environnement $\mathcal{E}$ de chaque bloc du système de contraintes en recalculant chacun de ces environnements de la même manière que celle décrite dans l'Algorithme 6.2 sauf que cette fois, seules les variables présentes dans des contraintes $Forge_c(E, \mathcal{K})$ de F peuvent avoir des valeurs de $\mathcal{E}$. Nous pouvons ainsi normaliser encore notre système de contraintes résultant de l'application des Règles [6.1] et [6.2], en utilisant la deuxième phase de la normalisation de la Définition 6.2.0.5. Nous désignons par $\downarrow^{Ext}$ l'application successive des Règles [6.1] et [6.2] (quand c'est possible), suivie à la fin par la phase 2 de normalisation. Cependant, la normalisation peut créer de nouvelles contraintes qui nécessitent d'être traitées encore une fois par les Règles [6.1] et [6.2]. Nous avons donc besoin d'itérer $\downarrow^{Ext}$. À ce stade, nous remarquons que :

Lemme 6.7.1.4 *$Patt(F) = Patt(F \downarrow^{Ext})$, et $[\![F]\!]^e_\emptyset = \left|\!\left[F \downarrow^{Ext} \right]\!\right|^e_\emptyset$.*

Preuve. La preuve découle directement de la définition des patterns et des Règles [6.1] et [6.2]. En effet, par définition, les patterns incluent les sous-termes modulo le même renommage des indices r et q. En outre, les Règles [6.1] et [6.2] extraient toujours des sous-termes et permettent ainsi de ne faire apparaître tout au long de la normalisation que des sous-termes de patterns. Plus précisément, $Patt(F)$ est stable par application de toute règle pour ces raisons :

- Les règles prioritaires ne créent pas de nouveaux patterns ;
- Les contraintes $Forge$, Sub ou encore les contraintes d'égalité génèrent de nouvelles contraintes avec uniquement des patterns hérités des contraintes précédentes ;
- Les règles d'entrelacement ne créent pas de nouveaux patterns puisqu'elles utilisent uniquement des termes existant auparavant dans F modulo des remplacements d'indices ;
- Les Règles [6.1] et [6.2] sont semblables à celles d'entrelacement. Elles ne génèrent pas de nouveaux patterns puisqu'elles utilisent les mêmes patterns du système de contraintes F avec des remplacements d'indices ;
- Les autres règles de formattage implicite telle que la mise en forme normale disjonctive n'affectent pas les termes de F et donc n'affectent pas les patterns.

Nous concluons que l'ensemble de patterns $Patt(F)$ d'un certain système de contrainte F est stable par notre système d'inférence étendu par les Règles [6.1] et [6.2]. D'où la preuve de la première partie de notre lemme, i.e. $Patt(F) = Patt(F \downarrow^{Ext})$.

Ensuite, nous avons montré en Section 6.4 que toutes les règles de notre système d'inférence sont correctes et complètes. En outre, les règles ajoutées (Règles [6.1] et [6.2]) sont exactement similaires aux règles d'entrelacement entre blocs, i.e. les règles du groupe G_6 de notre système d'inférence (voir Section 5.8.2 du Chapitre 5). Les Règles [6.1] et [6.2] conservent donc

la sémantique du système de contraintes vu qu'elles se basent sur une énumération exhaustive de toutes les valeurs possibles de X_m. Leurs preuves de correction et complétude sont similaires à celles présentées en Section 6.4, i.e la preuve de correction et complétude des Règles [5.34] et [5.35] du groupe G_6. D'où la preuve de la deuxième partie de notre lemme, i.e. $[\![F]\!]^e_\emptyset = \left[\!\left[F \downarrow^{Ext} \right]\!\right]^e_\emptyset$. □

En utilisant ce lemme et en respectant les hypothèses sur $\downarrow^{Ext}$, nous pouvons itérer $\downarrow^{Ext}$ jusqu'à obtenir un système de contraintes pour lequel aucune des Règles [6.1] ou [6.2] ne puisse être appliquée. Notons ici que, vu que les règles ajoutées (Règles [6.1] ou [6.2]) sont semblables à celles du groupe G_6 de notre système d'inférence, la preuve de terminaison donnée est aussi valide pour cette normalisation étendue. L'itération de $\downarrow^{Ext}$ à partir de F (le système de contraintes obtenu après la normalisation de l'Algorithme 6.2) termine donc en donnant comme résultat le système de contraintes F' qui se caractérise par quelques propriétés intéressantes, à savoir :

- Toute contrainte $(X_m = u)^\cdot$ se trouve avec $\bigwedge_{v \lhd u\bar{\delta}} \left(\forall k'. X_m \neq v\delta^{k'}_{q,m}\right)$ dans le même bloc ;
- Toute contrainte $(X_m \in Forge_c(E, \mathcal{K}))^\cdot$ se trouve avec $\bigwedge_{v \in Patt(F)} \left(\forall k'. X_m \neq v\delta^{k'}_{q,m}\right)$ dans le même bloc.

Proposition 6.7.1.5 *En itérant $\downarrow^{Ext}$ sur CBS_{k+1}, il existe e_{max} calculable tel que :*

$$\forall e \geq e_{max}, \quad [\![CBS_{k+1}]\!]^e_\emptyset = [\![CBS_{k+1}]\!]^{e_{max}}_\emptyset$$

Cette proposition permet de prouver directement le Lemme 6.2.0.9. La preuve de cette proposition fera le sujet de la Section 6.7.3. Cependant, cette section traite un système de contraintes F" qui résulte d'une autre transformation du système de contraintes fourni par la première étape F'. Cette deuxième transformation de F' à F" sera décrite en Section 6.7.2.

6.7.2 Deuxième étape : la transformation des contraintes $Forge_c$

Le but de la deuxième étape est de transformer le système de contraintes F' résultant de la première transformation en un autre système de contraintes F" où nous choisissons des valeurs pour toute variable sous une contrainte de type $Forge_c$. Les valeurs de ces variables sont choisies de telle manière qu'elles soient différentes de toutes les valeurs existantes dans les égalités de F'. En outre, nous choisissons une valeur différente pour chaque vecteur de variables.

Pour ce faire, soit c un terme clos de E_0 (connaissances initiales de l'intrus), i.e. un message fixé que l'intrus connaît dès le départ. Soit d tel que $d = \sum_{(X=u)^\cdot \in F'} dpth(u)$ la somme des profondeurs des égalités de F'. Notons que d est borné par la taille de F'. Pour tout $\overrightarrow{X} \in \overrightarrow{\mathcal{X}}$, nous choisissons un terme clos $b_{\overrightarrow{X}} = \langle c, .., c \rangle$ tel que $\forall \overrightarrow{X}, \overrightarrow{Y}$, $b_{\overrightarrow{X}} \neq b_{\overrightarrow{Y}}$. Nous définissons maintenant la valeur choisie $t_{\overrightarrow{X}}$ pour chaque vecteur de variable $\overrightarrow{X}$. Pour chaque $\overrightarrow{X} \in \overrightarrow{\mathcal{X}}$, soit $t_{\overrightarrow{X}} = \{..\{b_{\overrightarrow{X}}\}_c..\}_c$ le terme de profondeur $(d+1)$. Notons que, par construction, $\forall \overrightarrow{X}, \overrightarrow{Y}$, $t_{\overrightarrow{X}} \neq t_{\overrightarrow{Y}}$ et $\forall \overrightarrow{X}$, $t_{\overrightarrow{X}} \in Dy_c(E_0, \emptyset)$.

Nous pouvons maintenant définir le système de contraintes F" résultant de la clôture du système de contraintes F' (le résultat de la première transformation définie en Section 6.7.1) comme suit :

$$(X_i \in Forge_c(E))^\cdot \quad \longrightarrow \quad \left(X_i = t_{\overrightarrow{X}}\right)^\cdot \qquad [6.3]$$

Nous montrons ci-dessous que la Règle [6.3] est correcte et complète. Nous commençons tout d'abord par montrer la complétude de la Règle [6.3], i.e. que l'application de cette règle préserve la sémantique du système de contraintes. Ceci est prouvée par le Lemme 6.7.2.1 suivant :

Lemme 6.7.2.1 *Si* $\exists \sigma \in [\![F']\!]^e_\emptyset$ *alors* $\exists \sigma' \in [\![F'']\!]^e_\emptyset$.

La preuve de ce lemme est donnée ci-après. Ensuite, concernant la correction de la Règle [6.3], nous remarquons que, si $\sigma' \in [\![F'']\!]^e_\emptyset$, alors $\sigma' \in [\![F']\!]^e_\emptyset$. En effet, une contrainte $X_i = t_{\overrightarrow{X}}$ est plus restrictive qu'une contrainte $X_i \in Forge_c(E, \mathcal{K})$. Ainsi, nous avons la propriété suivante :

$$\big|[\![F']\!]\big|^e_\emptyset \neq \emptyset \text{ ssi } \big|[\![F'']\!]\big|^e_\emptyset \neq \emptyset$$

Nous concluons qu'il est suffisant de vérifier la satisfaisabilité de F" au lieu de la vérifier pour F'.

Complétude de la Règle [6.3]

Nous montrons dans ce paragraphe le Lemme 6.7.2.1 , i.e. que la Règle [6.3] est complète. Cependant, nous ne garantissons pas ici que toute solution de F' soit conservée pour F" mais plutôt qu'au moins une de ces solutions persiste dans F"'. En effet, soit $\sigma \in [\![F']\!]^e_\emptyset$ avec $F' = \forall Q \, \exists R \; B_1 \vee .. \vee B_p$. Nous définissons σ' telle que :

- $\forall X \in \mathcal{X}$, $\sigma'(X) = \sigma(X)$;
- $\forall \overrightarrow{X} \in \overrightarrow{\mathcal{X}}$, $\forall s \in \{1..e\}$, si $\exists u \in Patt(F')$, $\exists \tau$ avec $\tau(q) = s$ tel que $\sigma(X_s) = \sigma(u\tau)$ et u est minimal selon $\lhd$, alors $\sigma'(X_s) = \sigma'(u\tau)$;
  Notons que u est unique par minimalité selon $\lhd$ et vu que $Var^{\mathcal{X}}_{\mathcal{I}}(u) \subseteq \{q\}$.
- $\forall \overrightarrow{X} \in \overrightarrow{\mathcal{X}}$, $\forall s \in \{1..e\}$, si $\forall u \in Patt(F')$, $\forall \tau$ avec $\tau(q) = s$, nous avons $\sigma(X_s) \neq \sigma(u\tau)$, alors $\sigma'(X_s) = t_{\overrightarrow{X}}$.

L'idée derrière la définition de σ' est de conserver les valeurs attribuées aux variables non indicées puisque la Règle [6.3] n'affecte pas ces variables. Pour les valeurs des variables indicées, nous distinguons deux cas qui découlent des propriétés repérées à la fin de la Section 6.7.1. En effet, si une variable X_s existe dans une contrainte de type $Forge_c$ alors elle doit être remplacée par le terme $t_{\overrightarrow{X}}$. Cette contrainte est aussi accompagné par les contraintes négatives déclarant que cette variable doit être différente de tous les patterns. Si, par contre, la variable X_s est contrainte par une égalité avec un pattern donné (modulo remplacement d'indice), alors cette égalité est accompagnée par les contraintes négatives exprimant que cette variable doit être différente de tous les patterns inférieurs à sa valeur. Dans ce cas, la nouvelle valeur de X_s (par σ') doit être la nouvelle valeur de cette variable, i.e. $\sigma'(u\tau)$.

Nous montrons dans le reste de cette section que $\sigma' \in [\![F'']\!]^e_\emptyset$: grâce au fait que $\sigma \in [\![F']\!]^e_\emptyset$, nous savons que $\forall \tau_Q$, $\exists \tau \supseteq \tau_Q$, $\exists i \in \{1..p\}$, $\forall ctr \in B_i$ $\sigma \in [\![ctr]\!]^e_\tau$. Nous examinons tous les types de contraintes *ctr* possibles. Nous avons $\sigma \in [\![F']\!]^e_\tau$ et nous montrons que $\sigma' \in [\![F"]\!]^e_\tau$. D'après la définition d'un système de contraintes normalisé, une contrainte appartenant à ce système peut être soit une contrainte manipulant une variable non indicée, i.e. $(X_m = u)^\bullet$ ou $(X \in Forge_c(E, \mathcal{K}))$, soit une contrainte pour une variable indicée de type Forge, i.e. $(X_m \in Forge_c(E, \mathcal{K}))^\bullet$ ou de type égalité $(X_m = u)^\bullet$, soit enfin une contrainte négative, i.e $(\forall k \, X_m \neq u)$ ou bien $(X_m \notin Forge_c(E, \mathcal{K}))$. Dans ce qui suit, nous traitons ces différents types de contrainte, cas par cas.

1. Si $ctr = (X = u)^{sm}$ ou $ctr = (X \in Forge_c(E, \mathcal{K}))$, alors $\sigma' \in [\![ctr]\!]^e_\tau$ vu que $\sigma'(X) = \sigma(X)$;

2. Si $ctr = (X_m = u)^{\bullet}$, alors par définition de F', nous avons aussi dans le même bloc $\sigma \in \left[\!\!\left[\bigwedge_{v \lhd u\bar{\delta}} \left(\forall k'. X_m \neq v\delta^{k'}_{q,m} \right) \right]\!\!\right]^e_\tau$. Ainsi, nous avons un pattern $u' \in Patt(F')$ tel que $u' = u\bar{\delta}_q$ ou $u' = u\bar{\delta}_r$, avec $Var^{\mathcal{X}}_{\mathcal{I}}(u') \subseteq \{q\}$. En outre, u' est le pattern minimal par rapport à $\lhd$ (vu que $Var^{\mathcal{X}}_{\mathcal{I}}(u) \subseteq \{m\}$) tel que $\sigma(X_m\tau) = \sigma(u'\tau')$ avec $\tau'(q) = \tau(m)$, $\tau'(r) = \tau(k)$ et $k \in Var_{\mathcal{I}}(u) \backslash Var^{\mathcal{X}}_{\mathcal{I}}(u)$ si cet ensemble n'est pas vide. Nous concluons que u' suit le second point de la définition de σ', i.e. $\sigma'(X_m\tau) = \sigma'(u'\tau') = \sigma'(u\tau)$ et $\sigma' \in [\![ctr]\!]^e_\tau$.

3. Si $ctr = (X_m \in Forge_c(E, \mathcal{K}))^{\bullet}$, alors, par définition de F' nous avons : $\sigma \in \left[\!\!\left[\bigwedge_{v \in Patt(F)} \left(\forall k'. X_m \neq v\delta^{k'}_{q,m} \right) \right]\!\!\right]^e_\tau$. Ainsi, $\sigma'(X_m\tau) = t_{\overrightarrow{X}}$ et par conséquent, $\sigma' \in \left[\!\!\left[X_m = t_{\overrightarrow{X}} \right]\!\!\right]^e_\tau$.

4. Si $ctr = (\forall k\, X_m \neq u)$, alors nous prouvons par récurrence une propriété plus générale de σ et σ'. Cette propriété est effectuée en deux étapes. Dans la première étape, nous montrons que le terme choisi comme valeur d'une variable contrainte par un $Forge_c$ est toujours différente de tous les patterns. Cette étape est prouvée par la Prétention 6.7.2.2.

   Prétention 6.7.2.2 $\forall u \in Patt(F)$, $\forall \tau$, $\forall \overrightarrow{Z}$, $t_{\overrightarrow{Z}} \neq \sigma'(u\tau)$

   Preuve. La valeur d'une certaine variable indicée X_s pour σ' est un pattern utilisant uniquement des variables indicées par s. Ainsi, nous distinguons deux cas. Dans le premier cas, il existe un certain $t_{\overrightarrow{Y}}$ sous-terme de $\sigma'(u\tau)$. Ainsi, $dpth(\sigma'(u\tau)) > (d+1)$ vu que $u \notin \mathcal{X}_{\mathcal{I}}$. Dans le second cas, il n'existe aucun $t_{\overrightarrow{X}}$ sous-terme de $\sigma'(u\tau)$. Ainsi, $dpth(\sigma'(u\tau)) \leq d$ vu que les mêmes patterns ne peuvent pas apparaître deux fois dans la même branche de $\sigma'(u\tau)$ sans créer de cycle. Dans les deux cas $dpth(\sigma'(u\tau)) \neq (d+1)$, ce qui prouve la prétention. □

   Cette première étape sera utilisée dans la seconde étape pour montrer que, étant donné deux termes, l'un est un pattern et l'autre est un pattern modulo un remplacement d'indice, si ces deux termes sont inégaux modulo σ et τ alors ils restent inégaux modulo σ' et τ. Cette deuxième étape est prouvée en Prétention 6.7.2.3.

   Prétention 6.7.2.3 *$\forall w, v \in Patt(F) \cup \mathcal{X}_{\mathcal{I}}$ avec $u = w\delta_{r,r'}$, $\forall \tau$ si $\sigma(u\tau) \neq \sigma(v\tau)$ alors $\sigma'(u\tau) \neq \sigma'(v\tau)$.*

   Preuve. Supposons que la propriété est vraie pour $dpth(u) + dpth(v) < n$, pour un entier n. Soient u et v deux termes comme ci-dessus avec $dpth(u) + dpth(v) = n$. Nous distinguons plusieurs cas dépendant de la structure de u et v :

   – Si $u = f(\{u_i\})$ et $v = g(\{v_j\})$ alors $f \neq g$ et par la suite $\sigma'(u\tau) \neq \sigma'(v\tau)$, soit $f = g$ et $\exists i\ \sigma(u_i\tau) \neq \sigma(v_i\tau)$ avec $dpth(u_i) + dpth(v_i) < n$, i.e. $\sigma'(u_i\tau) \neq \sigma'(v_i\tau)$ et donc $\sigma'(u\tau) \neq \sigma'(v\tau)$. Ce cas inclut les constantes, i.e. f et g sans paramètres.
   – Si $u = Y_q$ et $v = Z_q$ avec $\sigma'(Y_q\tau) = \sigma'(u'\tau)$ et $\sigma'(Z_q\tau) = \sigma'(v'\tau)$, alors nous avons donc $\sigma(Y_q\tau) = \sigma(u'\tau)$ et $\sigma(Z_q\tau') = \sigma(v'\tau)$, avec $\sigma(u'\tau) \neq \sigma(v'\tau)$. Ainsi, d'une manière similaire au premier cas, nous avons $\sigma'(u'\tau) \neq \sigma'(v'\tau)$ et par conséquent $\sigma'(u\tau) \neq \sigma'(v\tau)$.

- Si $u = Y_q$ et $v = Z_q$ avec $\sigma'(Y_q\tau) = t_{\overrightarrow{Y}}$ et $\sigma'(Z_q\tau) = t_{\overrightarrow{Z}}$, alors $t_{\overrightarrow{Y}} \neq t_{\overrightarrow{Z}}$ et donc $\sigma'(Y_q\tau) \neq \sigma'(Z_q\tau)$. Notons que $u = v$ est impossible.
- Si $u = Y_q$ et $v \notin \mathcal{X}_\mathcal{I}$ avec $\sigma'(Y_q\tau) = t_{\overrightarrow{Y}}$, alors suite à la Prétention 6.7.2.2 ci-dessus, nous avons $\sigma'(Y_q\tau) = t_{\overrightarrow{Y}} \neq \sigma'(v\tau)$. Le même raisonnement est valide pour le cas inverse, i.e. $v = Y_q$ et $u \notin \mathcal{X}_\mathcal{I}$.
- Si $u = Y_q$ et $v = Z_q$ avec $\sigma'(Y_q\tau) = t_{\overrightarrow{Y}}$ et $\sigma'(Z_q\tau) = \sigma'(v'\tau)$, alors, suite à la Prétention 6.7.2.2, nous avons $\sigma'(Y_q\tau) = t_{\overrightarrow{Y}} \neq \sigma'(v'\tau)$.

Ceci prouve la prétention. □

En conséquence, vu que $ctr = (\forall k\, X_m \neq u)$ alors nous avons $\sigma(X_m\tau') \neq \sigma(u\tau')$ pour tout $\tau' \supseteq \tau$ tel que $Dom(\tau') = Dom(\tau) \cup \{k\}$, et par la suite, grâce à la Prétention 6.7.2.2, $\sigma'(X_m\tau') \neq \sigma'(u\tau')$ i.e. $\sigma' \in [\![\, ctr \,]\!]^e_\tau$.

5. Si $ctr = (X_m \notin Forge_c(E, \mathcal{K}))$, alors supposons $\overline{\mathcal{X}} = \{u \in \mathcal{X} \cup \mathcal{X}_\mathcal{I} \,|\, Var^\mathcal{X}_\mathcal{I}(u) \subseteq \{q\}\}$, et prouvons par itération deux propriétés génériques décrites en Prétention 6.7.2.4.

Prétention 6.7.2.4 *$\forall u \in Patt(F)$, $\forall \tau$, soit d une des dérivations minimales prouvant $\sigma'(u\tau) \in Dy(E'\sigma', \overline{\mathcal{K}}^e\tau\sigma')$ tel que aucun terme de $\{t_{\overrightarrow{X}}\}_{\overrightarrow{X} \in \overrightarrow{\mathcal{X}}}$ n'est décomposé, avec $E' = E \cup \{t_{\overrightarrow{X}}\}$. Alors, $\forall t$ connaissance dans d, $\exists v \in Patt(F) \cup \overline{\mathcal{X}}$ $\exists \tau'$ tel que $t = \sigma'(v\tau')$. En outre, remplacer chaque t par $\sigma(v\tau')$ et E' par $E" = E \cup \{\sigma(X_s) \,|\, \forall w \in Patt(F), \forall \tau', \sigma(X_s) \neq \sigma(w\tau')\}$ dans d maintient la validité de chaque application de règle.*

Preuve. Nous utilisons la structure de n'importe quelle dérivation minimale, i.e. tout terme décomposé est un sous-terme d'une connaissance initiale, et tout terme composé est un sous-terme soit d'un but, soit d'un message décomposé. Sinon, il y aurait des règles inutiles dans la dérivation. Avec ces constatations, nous distinguons deux cas selon que $\sigma'(w\tau')$ est composé ou décomposé pour un pattern ou une variable w :

- Soit $L = L_d(\sigma'(w\tau')) \in d$ telle que $w \in Patt(F) \cup \overline{\mathcal{X}}$. Soit t le terme généré par L. Nous distinguons deux cas. Pour le premier cas, $w = f(\{w_i\})$ et ainsi $\exists i$ tel que $t = \sigma'(w_i\tau')$, i.e. $\exists w' \in Patt(F) \cup \overline{\mathcal{X}}$ $\exists \tau"$ tel que $t = \sigma'(w'\tau")$, vu que les sous-termes des patterns sont aussi des patterns. Notons que $\tau" = \tau'$ sauf si f est un mpair ;
  Pour le deuxième cas, $w \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$, et ainsi $\forall \overrightarrow{X}$, $\sigma'(w\tau') \neq t_{\overrightarrow{X}}$ vu que ces termes ne peuvent pas être décomposées dans d. Par conséquent, $\exists v \in Patt(F)$ $\exists \tau"$ tel que $\sigma'(w\tau') = \sigma'(v\tau")$, et vu que $v \notin \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ nous pouvons refaire le même raisonnement qu'avec $w = f(\{w_i\})$.

- Soit $L = L_c(\sigma'(w\tau')) \in d$ tel que $w \in Patt(F) \cup \overline{\mathcal{X}}$. Soit $\{t_i\}$ l'ensemble de termes utilisés par L pour générer $\sigma'(w\tau')$. Nous distinguons alors deux cas selon que w est une variable ou non. Dans le premier cas, $w = f(\{w_i\})$, et donc $\forall i$ $\exists w'_i \in Patt(F) \cup \overline{\mathcal{X}}$ $\exists \tau"_i$ tel que $t_i = \sigma'(w'_i\tau"_i)$;
  Dans le deuxième cas, $w \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ avec $\exists v \in Patt(F)$ $\exists \tau"$ tel que $\sigma'(w\tau') = \sigma'(v\tau")$, et donc le même raisonnement que $w = f(\{w_i\})$ en suit ; Notons que $w \in \mathcal{X} \cup \mathcal{X}_\mathcal{I}$ avec $\exists \overrightarrow{X}$ tel que $\sigma'(w\tau') = t_{\overrightarrow{X}}$ est impossible par minimalité de la dérivation, vu que tout $t_{\overrightarrow{X}}$ est déjà dans E'.

Par conséquent, la première partie de la Prétention 6.7.2.4 suit par itération de ces deux étapes sur la structure de d, à partir de $E'\sigma'$ ou $\sigma'(u\tau)$. En outre, nous remarquons que tous les patterns choisis pour les connaissances dans d valident la propriété de sous-terme pour chaque règle, modulo un renommage des variables d'indices. En effet, chaque règle de décomposition génère un terme avec un pattern qui est un sous-terme du pattern de son argument, modulo renommage d'indice. La même constatation est valide pour les règles de composition. Ainsi, remplacer σ' par σ ne peut pas changer la validité de chaque application de règles, pourvu que les valeurs des variables sans patterns soient encore disponibles dans les connaissances initiales (i.e. E" remplaçant E'). La Prétention 6.7.2.4 est donc prouvé. □

La Prétention 6.7.2.4 est utilisé pour la contrainte $ctr = (X_m \notin Forge_c(E, \mathcal{K}))$, i.e. en supposant que $\sigma'(X_m\tau) \in Dy_c(E\sigma', \overline{\mathcal{K}}^e\tau\sigma')$ bien que $\sigma(X_m\tau) \notin Dy_c(E\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Alors, il existe une dérivation d définie comme dans la Prétention 6.7.2.4 avec $E' = E \cup \left\{t_{\overrightarrow{X}}\right\}$, vu que $\forall \overrightarrow{X}$ $t_{\overrightarrow{X}} \in Dy_c(E\sigma', \overline{\mathcal{K}}^e\tau\sigma')$, et ainsi, d admet les mêmes propriétés. Par conséquent, il existe une dérivation (valide) d' obtenue à partir de d comme celle décrite en Prétention 6.7.2.4, construisant $\sigma(u\tau)$ à partir de E"σ, et ayant la même structure que d, i.e. finissant par une règle de composition. Ensuite, E"$\setminus E$ contient uniquement des termes qui peuvent être construits par des dérivations (non-redondantes) en terminant par une règle de composition. Ainsi, il suit que $\sigma(X_m\tau) \in Dy_c(E\sigma, \overline{\mathcal{K}}^e\tau\sigma)$, ce qui est contradictoire au fait que $\sigma(X_m\tau) \notin Dy_c(E\sigma, \overline{\mathcal{K}}^e\tau\sigma)$. Par conséquent, notre hypothèse était fausse, i.e. $\sigma'(X_m\tau) \notin Dy_c(E\sigma', \overline{\mathcal{K}}^e\tau\sigma')$ et par la suite, $\sigma' \in [\![ctr]\!]^e_\tau$.

En conclusion, vu qu'il n'y a aucun autre type de contrainte dans un système de contraintes normalisé, notre Lemme 6.7.2.1 est donc prouvé, i.e. $\sigma' \in [\![F"]\!]^e_\emptyset$.

6.7.3 Troisième étape : l'existence d'une valeur e_{max} du paramètre n

Nous rappelons que nous sommes maintenant à la troisième étape de la preuve du Lemme 6.2.0.9. Nous rappelons aussi que nous avons à ce stade un système de contraintes F" issu de la deuxième transformation. Ce système de contraintes se caractérise par une propriété intéressante qui nous permettra de borner la valeur e du paramètre n tout en assurant l'équivalence de satisfaisabilité. Cette propriété est définie par la Propriété [6.1].

$$\forall \sigma \in [\![F"]\!]^e_\emptyset, \forall s \in \{1..e\}, \forall \overrightarrow{X} \in \overrightarrow{\mathcal{X}}, \quad \begin{array}{l} \text{soit } \sigma(X_s) = t_{\overrightarrow{X}} \\ \text{soit } \exists u \in Patt(F)\ \exists \tau \text{ avec } \tau(q) = s \text{ t.q. } \sigma(X_s) = \sigma(u\tau) \end{array} \quad [6.1]$$

Notons que cette structure découle des Règles [6.1], [6.2] et [6.3]. Dans tout ce qui suit, nous fixons une substitution close σ telle que $\sigma \in [\![F"]\!]^e_\emptyset$ en supposant qu'il existe une solution σ à F". Nous fixons aussi un certain pattern $u_{X_s} \in Patt(F) \cup \{t_{\overrightarrow{X}}\}$, et un remplacement d'indice τ_{X_s} tels que $\tau_{X_s}(q) = s$ et $\sigma(X_s) = \sigma(u_{X_s}\tau_{X_s})$ selon la Propriété [6.1]. Nous remarquons que, vu que $Patt(F)$ et $\overrightarrow{\mathcal{X}}$ sont finis, i.e. bornés par une fonction de la taille de la spécification du protocole, alors il existe nécessairement et seulement un nombre borné de choix possibles pour $\{u_{X_s}\}_{\overrightarrow{X} \in \overrightarrow{\mathcal{X}}}$, indépendamment de s. Soit donc e_1 cette borne. En outre, pour chaque X_s, il existe aussi seulement un nombre borné d'ensembles de connaissances E de F" et de clefs $\mathcal{K}$ tel que $\sigma(X_s) \in Dy_c(E\sigma, \mathcal{K}\sigma)$. Soit e_2 cette borne. Nous désignons par $e_{max} = e_1 \times e_2 \times \#\overrightarrow{\mathcal{X}}$. Nous remarquons qu'avec cette définition de e_{max}, si $e > e_{max}$ alors il existe $s' \neq s$ tel que

$\{u_{X_s}\}_{\overrightarrow{X}\in\overrightarrow{\mathcal{X}}} = \{u_{X_{s'}}\}_{\overrightarrow{X}\in\overrightarrow{\mathcal{X}}}$ et $\forall(\_ \in Forge_c(E,\mathcal{K}))$ dans F", $\forall\overrightarrow{X}$, $\sigma(X_s) \in Dy_c(E\sigma,\mathcal{K}\sigma)$ ssi $\sigma(X_{s'}) \in Dy_c(E\sigma,\mathcal{K}\sigma)$.

Nous utiliserons cette constatation pour prouver qu'il n'est pas nécessaire de chercher des attaques avec $e > e_{max}$. Nous prouvons donc le Lemme 6.7.3.1.

Lemme 6.7.3.1 *Si $e > e_{max}$ et $\sigma \in [\![F"]\!]^e_\emptyset$, alors $\exists\sigma' \in [\![F"]\!]^{e-1}_\emptyset$.*

PREUVE. Soient s et s' définis comme ci-dessus, i.e. tels que $\{u_{X_s}\}_{\overrightarrow{X}\in\overrightarrow{\mathcal{X}}} = \{u_{X_{s'}}\}_{\overrightarrow{X}\in\overrightarrow{\mathcal{X}}}$ et $\forall(\_ \in Forge_c(E,\mathcal{K}))$ in F", $\forall\overrightarrow{X}$, $\sigma(X_s) \in Dy_c(E\sigma,\mathcal{K}\sigma)$ ssi $\sigma(X_{s'}) \in Dy_c(E\sigma,\mathcal{K}\sigma)$. Nous remarquons que les égalités des patterns n'impliquent pas les égalités de leurs valeurs selon σ vu que les indices des constantes peuvent changer. Cependant, les égalités des patterns seraient suffisantes pour garantir qu'aux indices s et s', les variables valident les mêmes inégalités. Ainsi, nous allons enlever l'indice s' de σ en montrant que s peut effectivement remplacer s' dans tous les cas. Pour cela, soit γ le remplacement de tout c_v par c_{v-1} si $v > s'$, et de tout $c_{s'}$ par c_s, $c \in \overrightarrow{\mathcal{C}}$. On définit σ' comme suit :

- $\forall i < s', \forall\overrightarrow{X} \in \overrightarrow{\mathcal{X}}, \sigma'(X_i) = \gamma(\sigma(X_i))$;
- $\forall i \geq s', \forall\overrightarrow{X} \in \overrightarrow{\mathcal{X}}, \sigma'(X_i) = \gamma(\sigma(X_{i+1}))$;
- $\forall X \in \mathcal{X}, \sigma'(X) = \gamma(\sigma(X))$.

Supposons que notre système de contraintes soit $F" = \forall Q \exists R\, B_1 \vee .. \vee B_p$, et soit τ'_Q une substitution d'indices de Q vers $\{1..e-1\}$. Puis, soit τ_Q *la* substitution d'indices de Q vers $\{1..e\}$ telle que :

- $\forall i \in Q$, si $\tau'_Q(i) < s'$ alors $\tau_Q(i) = \tau'_Q(i)$, sinon $\tau_Q(i) = \tau'_Q(i) + 1$.

Intuitivement, nous insérons une colonne "blanche" à l'indice s'. Maintenant, par définition de σ $\exists\tau \supseteq \tau_Q$ tel que $\sigma \in [\![B_1 \vee .. \vee B_p]\!]^e_\tau$. Ainsi, nous définissons $\tau' \supseteq \tau'_Q$ tel que :

- $\forall r \in R$, si $\tau(r) < s'$ alors $\tau'(r) = \tau(r)$; si $\tau(r) = s'$ alors $\tau'(r) = s$; sinon $\tau'(r) = \tau(r) - 1$.

i.e. nous enlevons la colonne à l'indice s'. Nous devons prouver que $\sigma' \in [\![B_1 \vee .. \vee B_p]\!]^e_{\tau'}$. Cependant, $\exists i \in \{1..p\}$ tel que $\sigma \in [\![B_i]\!]^e_\tau$, et donc $\forall ctr \in B_i$, $\sigma \in [\![ctr]\!]^e_\tau$. Nous distinguons différents cas dépendant de ctr :

- Si $ctr = (X = u)^{sm}$ ou $ctr = (X \in Forge_c(E,\mathcal{K}))$, alors $\sigma' \in [\![ctr]\!]^e_{\tau'}$ vu que $\sigma'(X) = \sigma(X)$;
- Si $ctr = (X_m = u)^\bullet$, alors :
  - Si $\tau(m) \neq s'$ et $Var^{\mathcal{X}}_{\mathcal{I}}(u) = \{j\}$ avec $\tau(j) \neq s'$, alors $\sigma'(X_m\tau') = \gamma(\sigma(X_m\tau)) = \gamma(\sigma(u\tau)) = \sigma'(u\tau')$ vu que toute variable indicée dans u doit avoir m comme indice, et $\forall\overrightarrow{Y}$, $\sigma'(Y_m\tau') = \gamma(\sigma(Y_m\tau))$ vu que $\tau(m) \neq s'$; Ceci inclut $j = m$;
  - Si $\tau(m) = s'$ et $Var^{\mathcal{X}}_{\mathcal{I}}(u) = \{m\}$, alors $\sigma'(X_m\tau') = \gamma(\sigma(X_s))$. Vu que X_s et $X_{s'}$ partagent le même pattern pour σ, et vu que σ valide ctr, ce pattern est nécessairement u, i.e. $\gamma(\sigma(X_s)) = \gamma(\sigma(u[m \leftarrow s])) = \sigma'(u\tau')$;
  - Si $\tau(m) = s'$ et $Var^{\mathcal{X}}_{\mathcal{I}}(u) = \{j\} \neq \{m\}$, alors comme ci-dessus $\sigma'(X_m\tau') = \gamma(\sigma(X_s))$, et X_s et $X_{s'}$ partagent le même pattern u. Ainsi, $\sigma(X_s) = \sigma(u[j \leftarrow \tau(j)])$. Maintenant, soit $\tau(j) \neq s'$ et donc $\gamma(\sigma(u\tau)) = \sigma'(u\tau')$, ou $\tau(j) = s'$ et donc $\gamma(\sigma(u\tau)) = \sigma'(u[j \leftarrow s]) = \sigma'(u\tau')$. Dans les deux cas, $\sigma'(X_s) = \gamma(\sigma(X_s)) = \sigma'(u\tau')$, et donc, $\sigma' \in [\![ctr]\!]^e_{\tau'}$.

- Si $ctr = \left(X_m = t_{\overrightarrow{X}}\right)^{\bullet}$, alors $\sigma'(X_m\tau') = \gamma(\sigma(X_m\tau)) = \gamma(t_{\overrightarrow{X}}) = t_{\overrightarrow{X}}$ vu que X_s et $X_{s'}$ partagent le même pattern, et dans ce cas, ce pattern est $t_{\overrightarrow{X}}$. Ainsi, $\sigma' \in [\![ctr]\!]^e_{\tau'}$;
- Si $ctr = (\forall k\, X_m \neq u)$, alors :
  - Si $\tau(m) \neq s'$, alors $\forall v \in \{1..e\}$, $\sigma'(X_m\tau') = \gamma(\sigma(X_m\tau)) \neq \gamma(\sigma(u\tau[k \leftarrow v]))$, et donc $\forall v \in \{1..e-1\}$, $\sigma'(X_m\tau') \neq \sigma'(u\tau'[k \leftarrow v])$;
  - Si $\tau(m) = s'$, alors comme ci-dessus $\forall v \in \{1..e\}$, $\sigma(X_{s'}) = \sigma(X_m\tau) \neq \sigma(u\tau[k \leftarrow v])$. Cependant, X_s et $X_{s'}$ partagent le même pattern, et pour les deux, ce pattern est le minimal selon $\lhd$. Ainsi, u n'est pas le pattern de X_s, i.e. $\forall v \in \{1..e\}$, $\sigma(X_s) \neq \sigma(u\tau[k \leftarrow v])$. Il suit que $\forall v \in \{1..e-1\}$, $\sigma'(X_m\tau') \neq \sigma'(u\tau'[k \leftarrow v])$, i.e. $\sigma' \in [\![ctr]\!]^e_{\tau'}$.
- Si $ctr = (X_m \notin Forge_c(E, \mathcal{K}))$, alors, par hypothèse sur s et s' nous savons que $\sigma(X_s)$ et $\sigma(X_{s'})$ sont tous les deux dans $\in DY_c(E'\sigma, \mathcal{K}\sigma)$ ou aucun ne l'est, pour tout $(\_ \in Forge_c(E', \mathcal{K}))$ dans F", i.e. incluant E. Ainsi, $\sigma'(X_m\tau') \notin DY_c(E\sigma', \mathcal{K}\sigma')$ et $\sigma' \in [\![ctr]\!]^e_{\tau'}$.

Vu que ces types sont les seuls types de contraintes disponibles dans F", il suit que, comme prévu, $\sigma' \in [\![B_i]\!]^e_{\tau'}$. Par conséquent, pour tout τ'_Q de Q à $\{1..e-1\}$, nous pouvons trouver $\tau' \supseteq \tau'_Q$ de $Q \cup R$ à $\{1..e-1\}$ tel que $\sigma' \in [\![B_i]\!]^e_{\tau'}$, et ainsi, prouver que $\sigma' \in [\![F"]\!]^{e-1}_{\emptyset}$. Le Lemme 6.7.3.1 est ainsi prouvée. □

Ce lemme montre naturellement par itération que $[\![F"]\!]^e_{\emptyset}$ admet une solution avec $e > e_{max}$ si et seulement si $[\![F"]\!]^{e_{max}}_{\emptyset}$ admet une solution. Et vu que, pour tout e, $[\![F"]\!]^e_{\emptyset}$ admet une solution si et seulement si $[\![F]\!]^e_{\emptyset}$ admet une solution, ceci prouve la propriété.

6.8 Étude de cas

Nous avons montré grâce à toute la mécanique mise en place dans ce chapitre que le problème de l'insécurité de protocoles bien tagués avec clefs autonomes est décidable. La restriction concernant les clefs autonomes intervenait essentiellement pour prouver la terminaison de notre procédure. Nous avons donc testé deux scénarii de la version bien taguée du protocole Asokan-Ginzboorg. L'analyse de ces deux scénarii termine. Dans le premier scénario, nous avons considéré une exécution d'une seule session de ce protocole. Nous avons montré que l'application de notre procédure pour ce scénario n'a pas révélé d'attaque pour la propriété de secret de la clef du groupe. Une seule session de la version taguée du protocole de Asokan-Ginzboorg est donc sûre pour la propriété de secret de la clef du groupe. Dans le deuxième scénario, nous avons considéré deux sessions parallèles de la version taguée du protocole. Nous avons trouvé une attaque d'authentification.

6.8.1 Asokan-Ginzboorg à une seule session

Nous considérons une seule session de la version bien taguée du protocole Asokan-Ginzboorg donnée en Exemple 5.6.2.5. Nous considérons l'exécution suivante : $\langle (L,1),(S,1),(L,2),(S,2),(L,3)\rangle$. La propriété que nous voulons vérifier est le secret de la clef du groupe : $Sec = F(\langle mpair(z,(S_z)^z), s'\rangle)$. Nous avons une violation de la propriété de secret si l'intrus arrive à forger Sec à partir de toutes les connaissances qu'il a acquises durant l'exécution, d'où la dernière contrainte

$Sec \in Forge(E_5, \emptyset)$. Ainsi, le système de contraintes S que nous devons spécifier est le suivant :

(step 1) $mpair(i, \langle L, \{E_i\}_p \rangle) \in Forge(E_1, \emptyset)$
(step 2) $mpair(k, (\langle a_k, (\{\langle R_k, S_k \rangle\}_{(e)^k})^k \rangle)^k) \in Forge(E_2, \emptyset)$
(step 3) $mpair(q, (\{\langle mpair(u, (s_u)^u), S' \rangle\}_{(r_q)^q})^q) \in Forge(E_3, \emptyset)$
(step 4) $mpair(x, \langle a_x, \{\langle (S_x)^x, H(\langle mpair(z, (S_z)^z), s' \rangle) \rangle\}_{F(\langle mpair(z,(S_z)^z), s' \rangle)} \rangle) \in Forge(E_4, \emptyset)$
(step 5) $Sec \in Forge(E_5, \emptyset)$

où les ensembles E_i sont définis comme suit :

$$\begin{array}{lcl} E_1 & = & \{mpair(t, \langle l, \{e\}_p \rangle)\} \\ E_2 & = & E_1 \cup \{mpair(j, (\langle a_j, (\{\langle r_j, s_j \rangle\}_{(E_j)^j})^j \rangle)^j)\} \\ E_3 & = & E_2 \cup \{mpair(m, (\{\langle mpair(o, (S_o)^o), s' \rangle\}_{(R_m)^m})^m)\} \\ E_4 & = & E_3 \cup \{mpair(w, \langle a_w, \{\langle (s_w)^w, H(\langle mpair(y, (s_y)^y), S' \rangle) \rangle\}_{F(\langle mpair(y,(s_y)^y), S' \rangle)} \rangle)\} \\ E_5 & = & E_4 \end{array}$$

La normalisation de $S \backslash \{\text{step } 5\}$ par notre système de règles donne comme résultat le système de contraintes ci-dessous. Notons que le système résultant n'est pas mis en forme disjonctive pour éviter la redondance.

$$(S \backslash \{\text{step } 5\})\downarrow \; = \forall i,\, j,\, k,\, k_2,\, q,\, o,\, q_1,\, o_1,\, x,\, x_1,\, z,\, z_1,\, z_2,\, z_3,\, z_4,\, z_5$$
$$(\text{step } 1)\downarrow \wedge (\text{step } 2)\downarrow \wedge (\text{step } 3)\downarrow \wedge (\text{step } 4)\downarrow$$

avec,

$$(\text{step } 1)\downarrow \; = \left| \begin{array}{l} ((L \in Forge_c(E_1, \emptyset) \wedge (E_i = e)^m) \vee ((L = mpair(t, \langle l, \{e\}_p \rangle))^{sm} \wedge (E_i = e)^m) \\ \vee ((L = \langle l, \{e\}_p \rangle)^{sm} \wedge (E_i = e)^m) \vee ((L = l)^{sm} \wedge (E_i = e)^m) \\ \vee ((L = \{e\}_p)^{sm} \wedge (E_i = e)^m) \vee ((L = l)^{sm} \wedge (E_j = e)^m)) \end{array} \right.$$

$$(\text{step } 2)\downarrow \; = \left| \begin{array}{l} (((R_k = r_k)^m \wedge (S_k = s_k)^m \wedge (E_k = e)^f) \\ \vee ((R_{k_2} = r_{k_2})^m \wedge (S_{k_2} = s_{k_2})^m \wedge (E_{k_2} = e)^f)) \end{array} \right.$$

$$(\text{step } 3)\downarrow \; = \left| \begin{array}{l} (((R_q = r_q)^f \wedge (S' = s')^{sm} \wedge (S_o = s_o)^f) \\ \vee ((R_{q_1} = r_{q_1})^f \wedge (S' = s')^{sm} \wedge (S_{o_1} = s_{o_1})^f)) \end{array} \right.$$

$$(\text{step } 4)\downarrow \; = \left| \begin{array}{l} (((S_x = s_x)^f \wedge (S_{z_3} = s_{z_3})^f \wedge (S_{z_4} = s_{z_4})^f) \vee \\ ((S_x = s_x)^f \wedge (S_{z_1} = s_{z_1})^f \wedge (S_z = s_z)^f) \vee \\ ((S_{x_1} = s_{x_1})^f \wedge (S_{z_5} = s_{z_5})^f \wedge (S_{z_2} = s_{z_2})^f)) \end{array} \right.$$

Si nous essayons de normaliser la dernière étape, i.e. step 5, nous obtenons :

$$\begin{array}{l} \exists m \; \exists o_2 \; \exists w \\ (F(\langle mpair(z, (S_z)^z), s' \rangle) \in Sub_d(S', E_5, \mathcal{E}, \emptyset) \\ \wedge F(\langle mpair(y, (s_y)^y), S' \rangle) \in Forge(E_5, \{\{\langle (s_w)^w, H(\langle mpair(y, (s_y)^y), S' \rangle) \rangle\}_{F(\langle mpair(y,(s_y)^y), S' \rangle)}\})) \\ \vee (F(\langle mpair(z, (S_z)^z), s' \rangle) \in Sub_d((S_{o_2})^{o_2}, E_5, \mathcal{E}, \emptyset) \wedge \\ \qquad (R_m)^m \in Forge(E_5, \{\{\langle mpair(o, (S_o)^o), s' \rangle\}_{(R_m)^m}\})) \end{array}$$

Les entrelacements entre $F(\langle mpair(z, (S_z)^z), s' \rangle) \in Sub_d(S', E_5, \mathcal{E}, \emptyset)$ avec les contraintes sous-maîtres de S' de chacun des blocs obtenus ci-dessus mène à $\perp$ vu que dans chaque bloc, la contrainte sous-maître pour S' est $S' = s'$. De la même manière, l'entrelacement entre $F(\langle mpair(z, (S_z)^z), s' \rangle) \in Sub_d((S_{o_2})^{o_2}, E_5, \mathcal{E}, \emptyset)$ avec les contraintes maîtres de $\overrightarrow{S}$ de chacun des blocs obtenus ci-dessus mènent à $\perp$ vu que la valeur de S_o est une constante s_o. Ainsi, nous concluons qu'une exécution d'une seule session est sûre par rapport au secret de la clef du groupe.

6.8.2 Asokan-Ginzboorg à deux sessions en parallèle

Considérons deux sessions en parallèle de la version bien taguée du protocole Asokan Ginzboorg donnée en Exemple 5.6.2.5. Dans les spécifications suivantes, (A, i, j) désigne l'étape i du participant A dans la session j. En effet, $(L, \_, 1)$ (resp $(L, \_, 2)$) désigne le leader de la première session (resp. seconde session) et $(S, \_, 1)$ (resp $(S, \_, 2)$) désigne le simulateur de la première session (resp. seconde session).

La spécification de la première session est la suivante :

$$
\begin{array}{ll}
(L,1,1) & Init \Rightarrow mpair(t, \langle l, \{e\}_p \rangle) \\
(S,1,1) & mpair(i, \langle L, \{E_i\}_p \rangle) \Rightarrow mpair(j, (\langle a_j, (\{\langle r_j, s_j \rangle\}_{(E_j)^j})^j \rangle)^j) \\
(L,2,1) & mpair(k, (\langle a_k, (\{\langle R_k, S_k \rangle\}_{(e)^k})^k \rangle)^k) \Rightarrow mpair(m, (\{\langle mpair(o, (S_o)^o), s' \rangle\}_{(R_m)^m})^m) \\
 & \wedge \text{witness}((L, \_, 1), (S, \_, 1), f\_s, \langle mpair(o, (S_o)^o), s' \rangle) \\
(S,2,1) & mpair(q, (\{\langle mpair(u, (s_u)^u), S' \rangle\}_{(r_q)^q})^q) \Rightarrow \\
 & mpair(w, \langle a_w, \{\langle (s_w)^w, H(\langle mpair(y, (s_y)^y), S' \rangle) \rangle\}_{F(\langle mpair(y,(s_y)^y), S' \rangle)} \rangle) \\
 & \wedge \text{request}((S, \_, 1), (L, \_, 1), f\_s, \langle mpair(u, (s_u)^u), S' \rangle) \\
(L,3,1) & mpair(x, \langle a_x, \{\langle (S_x)^x, H(\langle mpair(z, (S_z)^z), s' \rangle) \rangle\}_{F(\langle mpair(z,(S_z)^z), s' \rangle)} \rangle) \Rightarrow End
\end{array}
$$

La spécification de la seconde session est la suivante :

$$
\begin{array}{ll}
(L,1,2) & Init \Rightarrow mpair(t, \langle l, \{e2\}_p \rangle) \\
(S,1,2) & mpair(i, \langle L', \{E'_i\}_p \rangle) \Rightarrow mpair(j, (\langle a_j, (\{\langle r'_j, s'_j \rangle\}_{(E'_j)^j})^j \rangle)^j) \\
(L,2,2) & mpair(k, (\langle a_k, (\{\langle R'_k, S'_k \rangle\}_{(e2)^k})^k \rangle)^k) \Rightarrow mpair(m, (\{\langle mpair(o, (S'_o)^o), s" \rangle\}_{(R'_m)^m})^m) \\
 & \wedge \text{witness}((L, \_, 2), (S, \_, 2), s\_s, \langle mpair(o, (S'_o)^o), s" \rangle) \\
(S,2,2) & mpair(q, (\{\langle mpair(u, (s'_u)^u), S" \rangle\}_{(r'_q)^q})^q) \Rightarrow \\
 & mpair(w, \langle a_w, \{\langle (s'_w)^w, H(\langle mpair(y, (s'_y)^y), S" \rangle) \rangle\}_{F(\langle mpair(y,(s'_y)^y), S" \rangle)} \rangle) \\
 & \wedge \text{request}((S, \_, 2), (L, \_, 2), s\_s, \langle mpair(u, (s'_u)^u), S" \rangle) \\
(L,3,2) & mpair(x, \langle a_x, \{\langle (S'_x)^x, H(\langle mpair(z, (S'_z)^z), s" \rangle) \rangle\}_{F(\langle mpair(z,(S'_z)^z), s" \rangle)} \rangle) \Rightarrow End
\end{array}
$$

Les étapes sont ordonnées comme suit : $W_L = 1, 2, 3$, $W_S = 1, 2$ avec $1 <_{W_L} 2$, $2 <_{W_L} 3$, et $1 <_{W_S} 2$ pour chaque session. Ensuite, la propriété que nous vérifions cette fois est l'authentification de la clef du groupe pour chacune des deux sessions. Elle est définie dans la spécification des deux sessions par les deux prédicats *witness* et *request*. En effet, pour la première session, nous avons $\text{witness}((L, \_, 1), (S, \_, 1), f\_s, \langle mpair(o, (S_o)^o), s' \rangle)$ pour dire que le leader de la première session veut partager la valeur de $\langle mpair(o, (S_o)^o), s' \rangle$ identifiée par f_s avec le simulateur de cette même session. Dans cette même session, nous avons aussi $\text{request}((S, \_, 1), (L, \_, 1), f\_s, \langle mpair(u, (s_u)^u), S' \rangle)$ pour dire que le simulateur de la session 1 accepte la valeur $\langle mpair(u, (s_u)^u), S' \rangle$ qui correspond à la valeur précédente puisqu'elle porte le même identifiant, ayant la garantie que le leader $(L, \_, 1)$ existe et qu'il est d'accord pour cette valeur. Nous procédons de la même manière pour la deuxième session pour spécifier la propriété d'authentification. Notons que, pour qu'une session soit sûre contre une attaque d'authentification, il faut que le *request* dans cette session ait la même valeur que celle du *witness* correspondant (même identifiant). Nous considérons l'exécution suivante :
$(L, 1, 1), (L, 1, 2), (S, 1, 1), (S, 1, 2), (L, 2, 1), (L, 2, 2), (S, 2, 1), (S, 2, 2), (L, 3, 1), (S, 3, 2)$.

Ainsi, le système de contraintes S que nous normalisons est le suivant :

$$
\begin{array}{ll}
(\text{step } 1) & mpair(i, \langle L, \{E_i\}_p \rangle) \in Forge(E_2, \emptyset) \\
(\text{step } 2) & mpair(i, \langle L', \{E'_i\}_p \rangle) \in Forge(E_3, \emptyset) \\
(\text{step } 3) & mpair(k, (\langle a_k, (\{\langle R_k, S_k \rangle\}_{(e)^k})^k \rangle)^k) \in Forge(E_4, \emptyset) \\
(\text{step } 4) & mpair(k, (\langle a_k, (\{\langle R'_k, S'_k \rangle\}_{(e2)^k})^k \rangle)^k) \in Forge(E_5, \emptyset) \\
(\text{step } 5) & mpair(q, (\{\langle mpair(u, (s_u)^u), S' \rangle\}_{(r_q)^q})^q) \in Forge(E_6, \emptyset) \\
(\text{step } 6) & mpair(q, (\{\langle mpair(u, (s'_u)^u), S" \rangle\}_{(r'_q)^q})^q) \in Forge(E_7, \emptyset) \\
(\text{step } 7) & mpair(x, \langle a_x, \{\langle (S_x)^x, H(\langle mpair(z, (S_z)^z), s' \rangle) \rangle\}_{F(\langle mpair(z,(S_z)^z), s' \rangle)} \rangle) \in Forge(E_8, \emptyset) \\
(\text{step } 8) & mpair(x, \langle a_x, \{\langle (S'_x)^x, H(\langle mpair(z, (S'_z)^z), s" \rangle) \rangle\}_{F(\langle mpair(z,(S'_z)^z), s" \rangle)} \rangle) \in Forge(E_9, \emptyset)
\end{array}
$$

où les ensembles E_i sont définis comme suit :

$$
\begin{array}{lcl}
E_2 & = & \{mpair(t, \langle l, \{e\}_p \rangle), mpair(t, \langle l2, \{e2\}_p \rangle)\} \\
E_3 & = & E_2 \cup \{mpair(j, (\langle a_j, (\{\langle r_j, s_j \rangle\}_{(E_j)^j})^j \rangle)^j)\} \\
E_4 & = & E_3 \cup \{mpair(j, (\langle a_j, (\{\langle r'_j, s'_j \rangle\}_{(E'_j)^j})^j \rangle)^j)\} \\
E_5 & = & E_4 \cup \{mpair(m, (\{\langle mpair(o, (S_o)^o), s' \rangle\}_{(R_m)^m})^m)\} \\
E_6 & = & E_5 \cup \{mpair(m, (\{\langle mpair(o, (S'_o)^o), s" \rangle\}_{(R'_m)^m})^m)\} \\
E_7 & = & E_6 \cup \{mpair(w, \langle a_w, \{\langle (s_w)^w, H(\langle mpair(y, (s_y)^y), S' \rangle) \rangle\}_{F(\langle mpair(y,(s_y)^y), S' \rangle)} \rangle)\} \\
E_8 & = & E_7 \cup \{mpair(w, \langle a_w, \{\langle (s'_w)^w, H(\langle mpair(y, (s'_y)^y), S" \rangle) \rangle\}_{F(\langle mpair(y,(s'_y)^y), S" \rangle)} \rangle)\} \\
E_9 & = & E_8
\end{array}
$$

Nous avons normalisé le système de contraintes S par notre système de règles d'inférence. Nous nous focalisons sur un seul bloc de notre système de contraintes normalisé :

$$
\begin{array}{l}
(L \in Forge_c(E_2, \emptyset) \wedge L' \in Forge_c(E_3, \emptyset) \wedge (E_{i_1} = e2)^m \wedge (E'_{i_4} = e)^m \wedge \\
(R_{k_1} = r'_{k_1})^m \wedge (S_{k_1} = s'_{k_1})^m \wedge (E'_{j_1} = e)^f \wedge \\
(R'_{k_2} = r_{k_2})^f \wedge (S'_{k_2} = s_{k_2})^f \wedge (E_{j_2} = e2)^f \wedge \\
(R'_{q_1} = r_{q_1})^f \wedge (S' = s")^{sm} \wedge (S'_{o_1} = s_{o_1})^f \wedge \\
(R_{q_2} = r'_{q_2})^f \wedge (S" = s')^{sm} \wedge (S_{o_2} = s'_{o_2})^f \wedge \\
(S_{x_1} = s'_{x_1})^f \wedge (S_{y_1} = s'_{y_1})^f \wedge (S'_{x_2} = s_{x_2})^f \wedge (S'_{z_1} = s_{z_1})^f \\
\wedge \text{witness}((L, \_, 1), (S, \_, 1), f\_s, \langle mpair(o, (S_o)^o), s' \rangle) \\
\wedge \text{witness}((L, \_, 2), (S, \_, 2), s\_s, \langle mpair(o, (S'_o)^o), s" \rangle) \\
\wedge \text{request}((S, \_, 1), (L, \_, 1), f\_s, \langle mpair(u, (s_u)^u), S' \rangle) \\
\wedge \text{request}((S, \_, 2), (L, \_, 2), s\_s, \langle mpair(u, (s'_u)^u), S" \rangle))
\end{array}
$$

Les variables d'indices de ce bloc sont quantifiées dans S comme suit :

$$\forall i_1, \forall i_4, \forall k_1, \forall k_2, \forall q_1, \forall q_2, \forall o_1, \forall o_2, \forall x_1, \forall x_2, \forall y_1, \forall z_1, \ \ \exists j_1, \exists j_2$$

Ce bloc correspond bien à une attaque d'authentification sur f_s. En effet, la solution σ de ce bloc donne s'_o comme valeur de S_o. Ainsi, le *request* pour f_s donne une valeur différente de la valeur du *witness* correspondant. Le même raisonnement est valable pour s_s.

6.9 Conclusion

Ce chapitre a constitué une suite du Chapitre 5 où nous avons présenté un modèle synchrone pour les protocoles de groupe et où nous avons défini notre système de contraintes adapté à

ce modèle. Pour celui-ci, nous avons introduit une classe bien particulière appelée la classe des protocoles bien tagués avec clefs autonomes. Nous avons également proposé un système de règles d'inférence manipulant notre système de contraintes.

Dans ce chapitre, nous avons proposé une procédure de vérification de cette classe de protocoles bien tagués avec clefs autonomes. Nous avons ensuite justifié le fait que notre modèle est une extension des modèles classiques en prouvant la terminaison dans le cas d'absence des variables d'indice et de *mpair*s. Nous avons par la suite montré la décidabilité des protocoles bien tagués avec clefs autonomes en prouvant la correction, la complétude, la terminaison et la satisfaisabilité du système normalisé de contraintes.

Notons que la restriction des clefs autonomes est essentiellement utilisée pour la preuve de terminaison. Cependant, par application de notre procédure pour l'exemple de Asokan-Ginzboorg tagué mais sans clefs autonomes, l'analyse de ce protocole termine et révèle une attaque d'authentification pour le cas de deux sessions en parallèle mais aussi prouve la sécurité de ce protocole pour le secret considérant une seule session. Ainsi, l'une de nos perspectives serait de considérer une classe plus large que celle avec clefs autonomes.

La procédure que nous proposons est une procédure de vérification d'une classe de protocoles considérant des listes paramétrées par leur longueur n. Quand notre procédure termine sans trouver de failles pour le protocole analysé, alors ce protocole n'admet aucune attaque pour toute valeur du paramètre n. Si nous appliquons ce raisonnement aux protocoles de groupe, paramétrés par leur nombre de participants, dans le cas d'absence d'attaques dans notre modèle un protocole analysé est alors prouvé comme sûr.

Nous rappelons ici que peu de travaux ont traité la vérification des protocoles de groupe en général. Quelques uns ([57]) ont considéré uniquement un intrus passif. D'autres n'autorisent que des clefs atomiques ([59, 60, 99]) et d'autres ([59, 99]) ne permettent pas d'effectuer les applications des listes (*list mapping*) . Quant à notre procédure, elle vérifie des protocoles de groupe et ceci en présence d'un intrus **actif**. Elle peut aussi modéliser les applications de listes et autorise les clefs composées.

Néanmoins, la classe de protocoles pour laquelle nous avons prouvé la décidabilité admet des messages tagués. Ce taguage a été introduit pour éviter des unifications à indices différents pouvant être exploitées pour coder un problème indécidable. D'autres travaux dans le domaine de la vérification de protocoles utilisaient des techniques de taguage pour les messages pour arriver à la décidabilité. Néanmoins, ces travaux ([14, 87]) ne considèrent pas les protocoles de groupe, ou plus généralement les protocoles considérant des listes paramétrées. En outre, dans notre cas, le taguage est limité aux messages contenants des variables indicées, i.e. des variables qui seraient instanciées par des messages de listes non bornées. Les autres messages n'ont pas besoin d'être tagués.

Règle	**Correction et Completude**	**Utilise**
Règle [5.2]	Proposition 6.4.2.1	
Règle [5.3]	Proposition 6.4.2.2	
Règle [5.9]	Correction : Proposition 6.4.2.6 Complétude : Proposition 6.4.2.3	Proposition 6.4.2.7 et Lemme 6.4.2.8 Lemmes 6.4.2.4 et 6.4.2.5.
Règles [5.10], [5.11] et [5.12]	Proposition 6.4.2.9	
Règle [5.13]	Proposition 6.4.2.11	Lemme 6.4.2.10
Règle [5.15]	Correction : Proposition 6.4.2.13 Complétude : Proposition 6.4.2.12	
Règle [5.16]	Proposition 6.4.2.14	
Règles [5.17] et [5.18]	Proposition 6.4.2.15	
Règle [5.19]	Proposition 6.4.2.16	
Règle [5.20]	Proposition 6.4.2.17	
Règle [5.23]	Proposition 6.4.2.18	
Règle [5.24]	Proposition 6.4.2.20	Lemme 6.4.2.19
Règle [5.25]	Proposition 6.4.2.21	
Règles [5.26] et [5.27]	Proposition 6.4.2.22	
Règles [5.28], [5.29] et [5.30]	Proposition 6.4.2.23	
Règle [5.31]	Proposition 6.4.2.24	
Règle [5.32]	Proposition 6.4.2.25	
Règle [5.33]	Proposition 6.4.2.26	
Règle [5.34]	Proposition 6.4.2.27	
Règle [5.35]	Proposition 6.4.2.28	

Tab. 6.1 – Synthèse des preuves de correction et de complétude

7

Conclusions et perspectives

La vérification des protocoles paramétrés et plus particulièrement des protocoles de groupe s'avère difficile pour trois raisons. La première est l'implication d'un nombre non borné, arbitraire de participants. Même une exécution normale de ces protocoles, i.e. sans intervention de l'intrus, génère un ensemble important de messages. La deuxième raison est la présence d'actions récursives/itératives effectuées par certains membres du groupe, tels que le serveur. Ces deux caractéristiques conduisent à l'indécidabilité du problème d'insécurité. La troisième difficulté des protocoles de groupe est la dynamique du groupe liée aux différents mouvements tels que l'ajout ou la sortie d'un ou plusieurs membres. Les propriétés de sécurité que doivent satisfaire ces protocoles sont alors plus sophistiquées.

Dans cette thèse, nous avons contribué à la vérification des protocoles paramétrés en nous focalisant sur les protocoles de groupe. Notre première contribution a été de retrouver expérimentalement des attaques sur quelques protocoles et d'en découvrir de nouvelles. Nous avons spécifié en HLPSL [25] certains protocoles de groupe afin de les vérifier par l'outil AVISPA [4].

Nos autres contributions sont des méthodes pouvant être automatisées afin d'assurer la vérification automatique de ces protocoles. Nous avons proposé dans un premier temps un modèle, appelé modèle de services pour les protocoles de groupe et leurs propriétés de sécurité. Nous avons ensuite traité la problématique du nombre non borné de participants des protocoles de groupe, et la vérification des protocoles cryptographiques avec listes paramétrées. Nous avons obtenu un résultat de décidabilité pour une classe de protocoles appelés protocoles bien tagués avec clefs autonomes.

7.1 Modélisation des propriétés de protocoles de groupe

Nous nous sommes intéressés dans un premier temps, à la recherche d'attaques sur les protocoles de groupe en tenant compte des différentes propriétés de sécurité. Nous avons défini un modèle, appelé modèle de services, inspiré du travail de Pereira et Quisquater [81], pour les protocoles de groupe et plus généralement les protocoles dits contribuants. Ce modèle permet de décrire ces protocoles, d'étudier leurs caractéristiques et leurs propriétés de sécurité afin de détecter de possibles attaques. Dans ce modèle, un protocole de groupe est défini comme un système formé de composantes (ensembles de termes) qui interagissent entre elles. Ces interactions doivent satisfaire certaines caractéristiques. Les propriétés de sécurité sont définies par combinaison de ces caractéristiques ou sous forme de relations entre les différentes composantes du système modélisant le protocole.

Le modèle de service a été utilisé d'abord pour identifier différents types d'attaques possibles sur des scénarii particuliers. L'application de ce modèle sur des protocoles tels que A-GDH.2 [8], SA-GDH.2 [8], Asokan-Ginzboorg [6] et Bresson-Chevassaut-Essiari-Pointcheval [22], a permis de détecter des types d'attaques possibles sur chacun.

La combinaison du modèle de service et de la résolution de contraintes nous a permis de proposer une stratégie de recherche d'attaques pour les protocoles de groupe. Une attaque est définie comme étant un système de contraintes que l'intrus doit résoudre en utilisant des règles d'inférence. Ce système de contraintes est composé d'un ensemble de contraintes provenant de la spécification du protocole et d'un autre ensemble de contraintes issu de la spécification de la ou des propriétés de sécurité considérées par le biais du modèle de service. La résolution de ce système de contraintes est modélisée sous forme d'un arbre de contraintes, que l'intrus construit au fur et à mesure, et qui correspondrait à la trace d'une attaque.

L'approche proposée a permis de retrouver d'anciennes attaques, d'en découvrir de nouvelles sur quelques protocoles de groupe tels que le protocole GDH.2 ou encore le protocole Asokan Ginzboorg. Les attaques trouvées pour les protocoles GDH.2 et A-GDH.2 ont été généralisées pour couvrir le cas de n participants.

7.2 Vérification de protocoles avec listes paramétrées

Nous nous sommes focalisé en second lieu sur la propriété du nombre non borné de participants qui caractérise les protocoles de groupe. Notre but est de chercher une classe décidable pour cette catégorie de protocoles en fixant le nombre de sessions considérées. Nous avons proposé une approche pour la vérification de protocoles paramétrés, i.e. qui impliquent un nombre non borné de participants, ou qui manipulent des listes paramétrées, et ce, pour un nombre fixe de sessions. Cela nous a conduit à définir un modèle *synchrone* pour une classe de protocoles, avec deux types de participants : un leader (ou serveur) et un certain nombre de participants *ordinaires* qui ont des comportements similaires vis-à-vis de la réception et de l'envoi des messages. L'idée était de projeter tous les participants ayant des comportements similaires en un seul participant *spécial* appelé *simulateur*. Mis à part les protocoles de groupe dont le paramètre est le nombre de participants impliqués, les listes paramétrées apparaissent aussi dans les protocoles de web services où les messages sont semi-structurés.

Le modèle proposé est une extension des modèles classiques tels que [89], qui se base sur la résolution de contraintes, et ce, afin de pouvoir traiter des listes de messages dont la longueur est donnée comme paramètre. Ces listes sont alors construites par un nouvel opérateur spécial nommé *mpair* qui désigne une liste construite sur le même patron. Nous avons montré que l'ajout naïf de l'opérateur *mpair* mène à l'indécidabilité du problème d'insécurité. D'où l'introduction de la classe de protocoles bien-tagués avec clefs autonomes.

Nous avons proposé une procédure de décision pour la classe de protocoles bien-tagués avec clefs autonomes, en présence d'un intrus **actif** et avec des clefs **composées**. En effet, nous avons tout d'abord présenté un ensemble de règles d'inférence démontrées correctes et complètes pour la classe de protocoles considérée. Nous avons prouvé ensuite que la normalisation d'un système de contraintes par notre ensemble de règles conduit à des contraintes en forme normale dont on peut tester la satisfaisabilité. Nous avons justifié que notre modèle est effectivement une extension des modèles classiques des protocoles cryptographiques en prouvant que, en cas d'absence d'indices ou de *mpair* pouvant générer des indices, nos règles d'inférence terminent. Finalement, nous avons montré que ces règles d'inférence terminent pour la classe des protocoles

bien tagués avec clefs autonomes, menant ainsi à un nouveau résultat de décidabilité pour la classe considérée.

Notons qu'en considérant des protocoles paramétrés dont le nombre de participants n n'est pas fixé, si notre procédure finit par ne pas trouver d'attaque, alors le protocole considéré n'admet aucune attaque pour toute valeur du paramètre n. Notons aussi que notre procédure permet aussi de traiter les protocoles avec **clefs composées**.

7.3 Perspectives

Notre contribution a porté sur deux axes : la modélisation des propriétés de sécurité des protocoles de groupe (voir Section 7.1), et le traitement du nombre non borné de participants (voir Section 7.2). Néanmoins, comme expliqué en Section 2.5, il existe un autre point très important relatif aux protocoles de groupe, qui est le traitement des agents malhonnêtes. Il serait donc intéressant d'étudier l'impact de ces agents malhonnêtes sur le déroulement d'un protocole de groupe. Nous distinguons dans ce qui suit, les perspectives relatives à la première contribution et celles relative à la deuxième.

7.3.1 Perspectives liées au modèle de services

Le modèle de service expliqué en Section 7.1 peut être complété de différentes manières :

- Le modèle de service que nous avons proposé a permis de formaliser des propriétés de sécurité en se basant sur des ensembles. Nous nous sommes focalisé sur les propriétés liées aux protocoles de groupe. Néanmoins, il existe d'autres propriétés intéressantes qui sont relatives aux protocoles qui sont proches de ceux de groupe mais, dont la définition est étroitement liée à la topologie du groupe. Nous citons comme exemple d'intérêt les protocoles hiérarchiques, qui sont couramment utilisés dans le cadre des applications militaires ou de forces civiles, ou simplement pour la gestion de clefs. Dans ce type de protocoles, le groupe est représenté sous forme de hiérarchie, i.e. de niveaux. Le secret d'une clef dépend du niveau des éléments détenant cette clef. Par exemple, une clef d'un niveau donné est connue des éléments de ce niveau ainsi que de tous les éléments des niveaux supérieurs. Il serait donc intéressant d'étendre notre modèle pour pouvoir traiter ces propriétés des protocoles hiérarchiques.
- Nous avons proposé dans un second lieu, un algorithme pour la recherche des attaques sur les protocoles de groupe. Nous avons pu retrouver et trouver de nouvelles attaques sur quelques protocoles. Néanmoins, toutes ces analyses étaient faites à la main. Ce travail doit donc être complété par une implantation afin de parvenir à un outil spécifique à la recherche d'attaque sur les protocoles de groupe traitant différentes propriétés.

7.3.2 Perspectives liées au résultat de décidabilité

Le résultat de décidabilité, expliqué en Section 7.2, obtenu pour la classe des protocoles bien tagués avec clefs autonomes peut être étendu de nombreuses manières :

- Nous avons prouvé le résultat de décidabilité pour la classe des protocoles bien tagués avec clefs autonomes. La restriction de clefs autonomes était utile pour la preuve de terminaison. Cependant, nous pensons que cette restriction peut être relâchée, en considérant les protocoles bien tagués mais avec une restriction moins forte que cette de clefs autonomes. Cette nouvelle restriction doit nous permettre de borner l'ensemble d'indices générés à partir des clefs.

– La signature que nous avons considérée dans notre deuxième contribution (voir Section 7.2), englobe les opérateurs de base (la paire, l'encryption symétrique et asymétrique, le hachage) et le nouvel opérateur de *mpair*. Cependant, certains protocoles de groupe utilisent les accords de clefs de Diffie-Hellman. Il serait donc intéressant d'ajouter des opérateurs algébriques tels que l'exponentiation ou le XOR afin d'élargir la classe de protocoles considérée.
– Les listes que nous avons considérée dans notre deuxième contribution sont toujours des listes de même longueur e. Plus particulièrement, en l'appliquant aux protocoles de groupe, cela suppose que, une fois que le nombre de participants est connu, ce nombre reste toujours le même. L'analyse exclut donc le cas de différentes sessions ayant différents nombres d'agents. Il serait donc intéressant de considérer des listes de longueurs différentes. Cet aspect pourrait être utilisé par exemple dans le traitement des mouvements de groupe tels que les entrées ou les sorties de membres.

7.3.3 Combinaison des deux modèles

Le modèle de services de la Section 7.1 a été essentiellement proposé pour formaliser les propriétés de sécurité qui caractérisent les protocoles de groupe et plus particulièrement les protocoles d'accord de clefs. Quant au deuxième modèle, résumé en Section 7.2, il a été proposé afin de gérer le nombre non borné non fixé des participants impliqués dans les protocoles de groupe et plus généralement les protocoles paramétrés. Cependant, ce dernier modèle focalise essentiellement sur la propriété de secret d'un message qui est généralement la clef de groupe ou une information pouvant dériver cette clef.

Il serait donc intéressant de combiner les deux travaux afin de fournir une plate-forme pour chercher d'éventuelles attaques sur les protocoles de groupe tout en considérant les différentes propriétés de sécurité qui caractérisent ce genre de protocoles. Dans ce cas, les propriétés ne sont plus spécifiées pour un modèle concret (instantiation du nombre de participants), mais plutôt d'une manière générique vu qu'elles seront relatives au simulateur et non pas aux différents participants. Cependant, cela nécessiterait l'introduction de relations entre les différents éléments des listes et donc des *mpair*.

7.3.4 Autres perspectives

Pouvoir gérer des listes paramétrées peut donner quelques applications dans d'autres domaines que la vérification de protocoles de groupe.

– Le deuxième modèle proposé dans ce manuscrit est appliqué pour un nombre borné de sessions. Cependant, le simulateur qui a été introduit pour simuler les actions d'un nombre non borné de participants peut aussi simuler un nombre non borné de sessions. Un message construit sur le nouvel opérateur introduit *mpair* pour le modèle synchrone représentait les informations relatives à chacun des participants simulés par le simulateur. Dans le cas de simulation d'un nombre non borné de sessions, un message construit sur l'opérateur *mpair* représenterait les informations relatives à chacune des sessions. La question qui dit se poser est alors la complétude de la transformation entre le modèle initial (avec nombre non borné de sessions) et le modèle synchrone (où on a simulé ce nombre de sessions par le biais de l'opérateur *mpair*), i.e. s'il y a une attaque dans le premier modèle, existe-t-il une attaque dans le second. Pour garantir cette complétude, des restrictions pourraient être introduites pouvant ainsi mener à une classe décidable avec nombre non borné de sessions.

- Nous nous sommes focalisés dans ce manuscrit sur les techniques de réécriture et de résolution de contraintes afin de modéliser le problème d'insécurité pour les protocoles de groupe. Cependant, il existe une autre technique, se basant, elle aussi, sur les modèles de traces qui contiennent généralement l'ensemble des messages échangés sur le réseau et les états locaux des participants ou les transitions effectuées. Cette technique est celle des clauses de Horn [37, 14]. Il serait intéressant de voir ce que peut donner une étude effectuée avec les clauses de Horn en ajoutant le nouvel opérateur *mpair*, surtout que certains des premiers travaux de décidabilité pour des classes des protocoles de groupe ont été trouvé en utilisant des classes de clauses de Horn [99, 59].

- Nous nous sommes focalisés durant toute la période de thèse aux protocoles de groupe et plus spécifiquement aux protocoles d'accord de clefs. Néanmoins, les deux modèles que nous avons proposé peuvent être étendus à d'autres domaines d'applications. Nous pouvons ainsi penser aux domaines suivants :
  - la vérification des protocoles de vente aux enchères [104] ou de vote électronique [51, 40]. Dans ce cadre, vu que notre modèle synchrone peut gérer le nombre non borné de participants, nous pouvons donc traiter le cas d'un ensemble de participants en train d'enchérir ou encore le cas d'un vote entre un ensemble de participants. Néanmoins, la question qui se pose est la formalisation des propriétés de sécurité qui caractérisent ces protocoles. Par exemple, nous trouvons la propriété d'anonymat qui exprime le fait qu'aucun participant, mis à par l'électeur lui même n'est capable de lier le vote à son électeur ; Les propriétés de sécurité des protocoles vote électronique sont proches de celles des protocoles de vente aux enchères.
  - la vérification des protocoles de signature de contrats tels que [48]. Vu que nous pouvons gérer les protocoles à nombre non borné de participants, nous pouvons donc traiter le cas de signature de contrat entre plusieurs participants. Cependant, le défit pour ce type de protocoles est tout d'abord les propriétés de sécurité qui les caractérisent. Comme exemple de propriétés, nous pouvons citer l'**équité**, i.e. il est impossible à un participant malhonnête d'obtenir un contrat valide sans permettre aux autres participants d'avoir le même. En outre, une fois qu'un participant honnête a obtenu une confirmation d'annulation (*abort*) du TTP (*Trusted Third Party*), il est impossible aux autres participants d'obtenir un contrat valide. Enfin, chaque participant honnête est capable de compléter le protocole.

Index

Bibliographie

[1] Security protocols open repository. `http://www.lsv.ens-cachan.fr/spore/`.

[2] Maude web site, 2000. `http://maude.csl.sri.com/`.

[3] Faulty group protocols, 2007. `http://homepages.inf.ed.ac.uk/gsteel/group-protocol-corpus/`.

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV*, volume 3576 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, 2005. Springer. `http://www.avispa-project.org`.

[5] A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In *FORTE '02 : Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 210–225, London, UK, 2002. Springer-Verlag.

[6] N. Asokan and P. Ginzboorg. Key Agreement in Ad hoc Networks. *Computer Communications*, 23(17) :1627–1637, 2000.

[7] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated Group Key Agreement and Friends. In *CCS '98 : Proceedings of the 5th ACM conference on Computer and Communications Security*, pages 17–26, New York, USA, 1998. ACM.

[8] G. Ateniese, M. Steiner, and G. Tsudik. New Multiparty Authentication Services and Key Agreement Protocols. *IEEE Journal on Selected Areas in Communications*, 18(4) :628–639, 2000.

[9] F. Bao, R. Deng, K. Nguyen, and V. Varadharajan. Multi-Party Fair Exchange with an Off-Line Trusted Neutral Party. In *DEXA '99 : Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, page 858, Washington, DC, USA, 1999. IEEE Computer Society.

[10] D. Basin, S. Mödersheim, and L. Viganò. OFMC : A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3) :181–208, 2005.

[11] K. Becker and U. Wille. Communication complexity of group key distribution. In *CCS'98 : Proceedings of the 5th ACM conference on Computer and communications security*, pages 1–6, New York, NY, USA, 1998. ACM.

[12] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[13] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–, Berkeley, CA, USA, May 2004. IEEE Computer Society.

[14] B. Blanchet and A. Podelski. Verification of Cryptographic Protocols : Tagging Enforces Termination. In A. Gordon, editor, *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *Lecture Notes on Computer Science*, pages 136–152, Warsaw, Poland, 2003. Springer Verlag.

[15] J. M. Bohli, M. Vasco, and R. Steinwandt. Secure Group Key Establishment Revisited. Cryptology ePrint Archive, Report 2005/395, 2005. `http://eprint.iacr.org/`.

[16] M. Bouallagui, Y. Chevalier, M. Rusinowitch, M. Turuani, and L. Vigneron. Analyse Automatique de Protocoles de Sécurité avec CASRUL. In *Actes de SAR'2002, Sécurité et Architecture Réseaux*, Juillet 2002. `http://www.loria.fr/~vigneron/Work/papers/BouallaguiCRTV-SAR02.ps.gz`.

[17] M. S. Bouassida, N. Chridi, I. Chrisment, O. Festor, and L. Vigneron. Automatic Verification of a Key Management Architecture for Hierarchical Group Protocols. In F. Cuppens and H. Debar, editors, *Proceedings of 5th Conference on Security and Network Architectures (SAR)*, pages 381–397, Seignosse, France, June 2006.

[18] M.S. Bouassida, N. Chridi, I. Chrisment, O. Festor, and L. Vigneron. Automated Verification of a Key Management Architecture for Hierarchical Group Protocols. *Annals of Telecommunications*, 62(11-12) :1365–1387, November-December 2007.

[19] M.S. Bouassida, N. Chridi, I. Chrisment, B. Fontan, S. Mota, P. Owezarski, H. Ragab Hassan, P. , De Saqui Sannes, and T. Villemur. L2.5 - Spécification du système global, intégration des services de sécurité au protocole de gestion de clés-. Contrat RNRT SAFECAST, LAAS-LORIA, Novembre 2005.

[20] C. Boyd. On Key Agreement and Conference Key Agreement. In *ACISP'97 : Proceedings of the Second Australasian Conference on Information Security and Privacy*, pages 294–302, London, UK, 1997. Springer-Verlag.

[21] L. Bozga, Y. Lakhnech, and M. Périn. HERMES : An Automatic Tool for Verification of Secrecy in Security Protocols. In *Computer Aided Verification, 15th International Conference (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 219–222, Boulder, CO, USA, July 2003. Springer.

[22] E. Bresson, O. Chevassut, A. Essiari, and D. Pointcheval. Mutual Authentication and Group Key Agreement for Low-Power Mobile Devices. *Journal of Computer Communications*, 27(17) :1730–1737, July 2004. Special Issue on Security and Performance in Wireless and Mobile Networks. Elsevier Science.

[23] J. Bull and D. Otway. The authentication protocol. Technical report, DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/0.3, Defence Research Agency, Malvern, UK, 1997.

[24] Y. Chevalier. *Résolution de problèmes d'accessibilité pour la compilation et la validation de protocoles cryptographiques.* Thèse de doctorat, Université Henri Poincaré, Nancy, Décembre 2003.

[25] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Proceedings of Workshop on Specification and Automated Processing of Security Requirements (SAPS)*, Linz, Austria, September 2004. `http://www.loria.fr/~vigneron/Work/papers/ChevalierCCHDMMV-SAPS04.pdf`.

[26] N. Chridi, B. Fontan, and S. Mota. L2.5 SAFECAST : Spécification du système global - Intégration des services de sécurité au protocole de gestion de clés -. Contrat RNRT SAFECAST, LAAS-LORIA, 2005. `http://hal.inria.fr/inria-00000789/en/`.

[27] N. Chridi, M. Turuani, and M. Rusinowitch. Constraints-based Verification of Parameterized Cryptographic Protocols. Research Report RR-6712, INRIA, 2008. `http://hal.inria.fr/inria-00336539/en/`.

[28] N. Chridi, M. Turuani, and M. Rusinowitch. Towards a Constrained-based Verification of Parameterized Cryptographic Protocols. In M. Hanus, editor, *LOPSTR 2008 : Logic-based Program Synthesis and Transformation*, Valencia, Spain, 2008. `http://hal.inria.fr/inria-00332484/en/`.

[29] N. Chridi, M. Turuani, and M. Rusinowitch. Decidable Analysis for a Class of Cryptographic Group Porotocols with Unbounded Lists. In *CSF 2009 : 22nd IEEE Computer Security Foundations Symposium 2009*, pages 277–289, Port Jefferson, New York, USA, July 2009. IEEE Computer Society.

[30] N. Chridi and L. Vigneron. Modélisation des propriétés de sécurité de protocoles de groupe. In *Actes du 1er Colloque sur les Risques et la Sécurité d'Internet et des Systèmes, CRISIS*, pages 119–132, Bourges, France, Octobre 2005. Élu meilleur papier de la conférence.

[31] N. Chridi and L. Vigneron. Sécurité des communications de groupe. *Revue de l'électricité et de l'électronique*, (6/7) :51–60, juin/juillet 2006.

[32] N. Chridi and L. Vigneron. Strategy for Flaws Detection based on a Services-driven Model for Group Protocols. In B. Blanc, A. Gotlieb, and C. Michel, editors, *Proceedings of the 1st Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA*, pages 88–99, Nantes, France, September 2006.

[33] N. Chridi and L. Vigneron. *Strategy for Flaws Detection based on a Services-driven Model for Group Protocols*, chapter 24, pages 361–370. Future and Trends in Constraint Programming. ISTE, April 2007.

[34] J. Clark and J. Jacob. A survey of authentication protocol literature. Technical Report 1.0, 1997. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.3372`.

[35] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4) :443–487, 2000.

[36] H. Comon and R. Nieuwenhuis. Induction=I-Axiomatization+First-Order Consistency. *Information and Computation*, 159(1–2) :151–186, 2000.

[37] H. Comon-Lundh and V. Cortier. New Decidability Results for Fragments of First-Order Logic and Application to Cryptographic Protocols. In *14th International Conference on Rewriting Techniques and Applications (RTA'03)*, pages 148–164, Valencia, Spain, June 2003.

[38] H. Comon-Lundh and V. Cortier. Tree automata with one memory set constraints and cryptographic protocols. *Theoretical Computer Science*, 331(1) :143–214, 2005.

[39] V. Cortier. Vérifier les protocoles cryptographiques. *Technique et Science Informatiques*, 24(1) :115–140, 2005.

[40] R. J. F. Cramer, M. Franklin, L. A. M. Schoenmakers, and M. Yung. Multi-authority secret-ballot elections with linear work. Technical Report CS-R9571, Amsterdam, The Netherlands, 1995.

[41] C. J. F. Cremers. The Scyther Tool : Verification, Falsification, and Analysis of Security Protocols. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418, Princeton, NJ, USA, July 2008. Springer.

[42] S. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Standford University, Stanford, CA, USA, 1992.

[43] G. Denker and J. Millen. CAPSL Integrated Protocol Environment. In *DARPA Information and Survivability Conference and Exposition, DISCEX*, pages 207–221, Hilton Head, SC, 2000. IEEE Computer Society. `http://www.sdl.sri.com/papers/denmil00/`.

[44] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6) :644–654, 1976.

[45] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2) :198–207, 1983.

[46] B. Donovan, P. Norris, and G. Lowe. Analyzing a Library of Security Protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[47] N. Durgin, P. Lincoln, J. Mitchell, and A Scedrov. Undecidability of Bounded Security Protocols. In *Workshop on Formal Methods and Security Protocols*, Trento, Italy, July 1999.

[48] J. Garay and P. MacKenzie. Abuse-Free Multi-party Contract Signing. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 151–165, London, UK, 1999. Springer-Verlag.

[49] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In D. A. McAllester, editor, *CADE-17 : Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290, Pittsburgh, PA, USA, June 2000. Springer.

[50] T. Genet and V. Tong. Reachability Analysis of Term Rewriting Systems with Timbuk. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706, Havana, Cuba, December 2001. Springer.

[51] M. Hirt and K. Sako. Efficient Receipt-Free Voting Based on Homomorphic Encryption. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 539–556, Bruges, Belgium, May 2000. Springer Berlin / Heidelberg.

[52] A. Huima. Efficient infinite-state analysis of security protocols. In *Proceeding of the Workshop on Formal Methods and Security Protocols (FLOC'99)*, 1999.

[53] D. Jackson. Alloy : a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2) :256–290, April 2002.

[54] J. Katz and J. S. Shin. Modeling insider attacks on group key-exchange protocols. In *CCS '05 : Proceedings of the 12th ACM conference on Computer and communications security*, pages 180–189, New York, NY, USA, 2005. ACM.

[55] Y. Kim, A. Perrig, and G. Tsudik. Communication-efficient group key agreement. In *Sec '01 : Proceedings of the 16th international conference on Information security : Trusted information*, pages 229–244, 2001.

[56] Y. Kim, A. Perrig, and G. Tsudik. Tree-based group key agreement. *ACM Transactions on Information and System Security (TISSEC)*, 7(1) :60–96, 2004.

[57] S. Kremer, A. Mercier, and R. Treinen. Proving Group Protocols Secure Against Eavesdroppers. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 116–131, Sydney, Australia, 2008. Springer-Verlag.

[58] K. Kürtz, R. Küsters, and T. Wilke. Selecting theories and nonce generation for recursive protocols. In *FMSE '07 : Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 61–70, New York, NY, USA, 2007. ACM.

[59] R. Küsters and T. Truderung. On the Automatic Analysis of Recursive Security Protocols with XOR. In *Proceedings of the 24th Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, volume 4393 of *Lecture Notes in Computer Science*, pages 646–657. Springer, 2007.

[60] R. Küsters and T. Wilke. Automata-based Analysis of Recursive Cryptographic Protocols. In V. Diekert and M. Habib, editors, *21st Symposium on Theoretical Aspects of Computer Science (STACS 2004)*, volume 2996 of *Lecture Notes in Computer Science*, pages 382–393, Le Corum, Montpellier, France, March 2004. Springer-Verlag.

[61] G. Lowe. Breaking and fixing the Needham-Shroeder public-key protocol using FDR. In M. Tiziana and Bernhard S., editors, *TACAS'96 : Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, Passau, Germany, March 1996. Springer.

[62] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSF)*, pages 96–105, Rockport, Massachusetts, USA, June 1998. IEEE Computer Society.

[63] M. Manulis. Survey on Security Requirements and Models for Group Key Exchange. Technical Report 2006/02, Horst-Görtz Institute, Network and Data Security Group, 2006.

[64] C. Meadows. The NRL Protocol Analyzer : An Overview. *Journal of Logic Programming*, 26(2) :113–131, 1996.

[65] C. Meadows. Extending Formal Cryptographic Protocol Analysis Techniques for Group Protocols and Low-Level Cryptographic Primitives. In P. Degano, editor, *Proceedings of the 1st Workshop on Issues in the Theory of Security, WITS*, pages 87–92, Geneva, Switzerland, 2000.

[66] C. Meadows and P. Syverson. Formalizing GDOI group key management requirements in NPATRL. In *Proceedings of the 8th ACM conference on Computer and Communications Security, CCS*, pages 235–244, New York, NY, USA, 2001. ACM.

[67] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*, chapter Key Establishment Protocols. CRC Press, 1996. `http://www.cacr.math.uwaterloo.ca/hac/`.

[68] J. Millen and G. Denker. MuCAPSL. *DARPA Information Survivability Conference and Exposition*, 1 :238, 2003.

[69] J. Millen and V. Shmatikov. Constraint Solving for Bounded-Process Cryptographic Protocol Analysis. In *CCS '01 : Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 166–175, New York, NY, USA, 2001. ACM.

[70] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. In *SP'97 : Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 141, Washington, DC, USA, 1997. IEEE Computer Society.

[71] S. Mittra. Iolus : A Framework for Scalable Secure Multicasting. In *Proceedings of the ACM SIGCOMM'97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 277–288, Cannes, France, September 1997.

[72] D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2) :177–202, 2003.

[73] J. Nam, S. Kim, and D. Won. Attacks on Bresson-Chevassut-Essiari-Pointcheval's Group Key Agreement Scheme for Low-Power Mobile Devices. Cryptology ePrint Archive, Report 2004/251, 2004. `http://eprint.iacr.org/`.

[74] J. Nam, S. Kim, and D. Won. A Weakness in the Bresson-Chevassut-Essiari-Pointcheval's Group Key Agreement Scheme for Low-Power Mobile Devices. *IEEE Communication Letters*, 9(5) :429–431, 2005.

[75] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM (CACM)*, 21(12) :993–999, 1978.

[76] L. C. Paulson. *Isabelle : a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[77] L. C. Paulson. Mechanized Proofs for a Recursive Authentication Protocol. In *10th Computer Security Foundations Workshop (CSFW)*, pages 84–95. IEEE Computer Society Press, 1997.

[78] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2) :85–128, 1998.

[79] O. Pereira. *Modelling and Security Analysis of Authenticated Group Key Agreement Protocols*. PhD thesis, Université catholique de Louvain, May 2003.

[80] O. Pereira and J.-J. Quisquater. Security Analysis of the Cliques Protocols Suites : First Results. In *IFIP/Sec '01 : Proceedings of the IFIP TC11 Sixteenth Annual Working Conference on Information Security*, pages 151–166, Deventer, The Netherlands, 2001. Kluwer.

[81] O. Pereira and J.-J. Quisquater. Some Attacks Upon Authenticated Group Key Agreement Protocols. *Journal of Computer Security*, 11(4) :555–580, 2003.

[82] O. Pereira and J.-J. Quisquater. Generic Insecurity of Cliques-Type Authenticated Group Key Agreement Protocols. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17)*, pages 16–19, Pacific Grove, CA, USA, June 2004. IEEE Computer Society.

[83] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3) :115–116, 1981.

[84] A. Pettorossi, M. Proietti, and V. Senni. Transformational Verification of Parameterized Protocols Using Array Formulas. In P. M. Hill, editor, *Logic Based Program Synthesis and Transformation, 15th International Symposium, LOPSTR 2005,* , volume 3901 of *Lecture Notes in Computer Science*, pages 23–43, London, UK, September 2005. Springer.

[85] S. Rafaeli and D. Hutchison. A survey of key management for secure group communication. *ACM Computing Surveys*, 35(3) :309–329, September 2003.

[86] H. Ragab Hassan, A. Bouabdallah, H. Bettahar, and Y. Challal. HI-KD : Hash-based hierarchical key distribution for group communication. In *IEEE-INFOCOM'05*, March 2005. short paper.

[87] R. Ramanujam and S. Suresh. Tagging Makes Secrecy Decidable with Unbounded Nonces as Well. In *FST TCS 2003 : Foundations of Software Technology and Theoretical Computer*

Science, 23rd Conference, Mumbai, India, volume 2914 of *Lecture Notes in Computer Science*, pages 363–374. Springer, 2003.

[88] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[89] M. Rusinowitch and M. Turuani. Protocol insecurity with a finite number of sessions, composed keys is NP-complete. *Theoretical Computer Science*, 1-3(299) :451–475, 2003.

[90] P. Ryan and S. Schneider. *The modelling and analysis of security protocols : the csp approach*. Addison-Wesley Professional, 2000.

[91] A. T. Sherman and D. A. McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering (TSE)*, 29(5) :444–458, 2003.

[92] D. X. Song, S. Berezin, and A. Perrig. Athena : A Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9(1/2) :47–74, 2001.

[93] G. Steel and A. Bundy. Attacking Group Multicast Key Management Protocols using CORAL. In A. Armando and L. Viganó, editors, *Proceedings of the Workshop on Automated Reasoning for Security Protocol Analysis, ARSPA*, volume 125 :1 of *Electronic Notes in Theoretical Computer Science*, pages 125–144, 2004.

[94] G. Steel, A. Bundy, and M. Maidl. Attacking a Protocol for Group Key Agreement by Refuting Incorrect Inductive Conjectures. In D. Basin and M. Rusinowitch, editors, *Proceedings of 2nd Int. Joint Conference on Automated Reasoning, IJCAR*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 137–151, Cork, Ireland, 2004. Springer.

[95] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES : A new approach to group key agreement. In *Proceedings of the 18th International Conference on Distributed Computing Systems, ICDCS*, pages 380–387, Amsterdam, The Netherlands, 1998. IEEE Computer Society.

[96] M. Taghdiri and D. Jackson. A Lightweight Formal Analysis of a Multicast Key Management Scheme. In *Formal Techniques for Networked and Distributed Systems, FORTE*, volume 2767 of *Lecture Notes in Computer Science*, pages 240–256, Berlin, Germany, 2003. Springer.

[97] S. Tanaka and F. Sato. A Key Distribution and Rekeying Framework with Totally Ordered Multicast Protocols. In *ICOIN '01 : Proceedings of the The 15th International Conference on Information Networking*, pages 831–838, Washington, DC, USA, 2001. IEEE Computer Society.

[98] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand Spaces : Proving Security Protocols Correct. *Journal of Computer Security*, 7(2/3) :191–230, 1999.

[99] T. Truderung. Selecting theories and recursive protocols. In M. Abadi and L. Alfaro, editors, *Concurrency Theory, 16th International Conference, CONCUR 2005*, volume 3653 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2005.

[100] G. Tsudik. Secure and Minimal Protocols for Authenticated Key Distribution. *Computer Communications Journal*, 18 :645–653, 1995.

[101] M. Turuani. *Sécurité des protocoles cryptographiques : décidabilité et complexité*. PhD thesis, Université Henri Poincaré, Décembre 2003.

[102] M. Turuani. The CL-Atse Protocol Analyser. In *Term Rewriting and Applications - Proc. of RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286, Seattle, WA, USA, 2006.

[103] C. Weidenbach. Towards an Automatic Analysis of Security Protocols in First-Order Logic. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328, Trento, Italy, July 1999. Springer.

[104] E. Wolfstetter. Auctions : An Introduction. *Journal of Economic Surveys*, 10(4) :367–420, December 1996. `http://ideas.repec.org/a/bla/jecsur/v10y1996i4p367-420.html`.

[105] C. Wong, M. Gouda, and S. Lam. Secure Group Communications Using Key Graphs. In *Proceedings of the ACM SIGCOMM'98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 68–79, Vancouver, B.C., Canada, 1998.

[106] J. XU. *Automatic Verification of Parameterized Systems*. PhD thesis, Department of Computer Science, new york University, New York, NY, USA, 2005. Adviser-Pnueli, Amir.

A

Spécification en HLPSL des protocoles analysés par cl-atse

A.1 Spécification en HLPSL du protocole Asokan

A.1.1 Le protocole Asokan-Ginzboorg en général

```
%% the asokan ginzboorg with keys for participants
%% we have two roles: a leader L and a member M

%% First Role: a member of a group
role member (L,M: agent,
           Members : (agent) set,
   P: symmetric_key,
   F,H: hash_func,
           Snd, Rcv: channel(dy))
     played_by M def=
   local State: nat,
         E: public_key,
         R: symmetric_key,
         S: nat,
         GroupMembers: (agent) set,
         X: message
   const id1: protocol_id
   init State:=0 /\ GroupMembers := Members
   transition
     step1. State=0 /\ Rcv(L.{E'}_P)
       =|> R' := new()
           /\ S' := new()
           /\ Snd(M.{R'.S'}_E')
           /\ State':=1

     step2. State=1 /\ Rcv({X'}_R)
       =|> State':=2
           /\ GroupMembers' := cons(L,GroupMembers)
```

```
           /\ Snd(M.{S.H(X')}_F(X'))
           /\ secret(F(X'),id1,{GroupMembers'})
end role

%% Second Role: The Leader of the group
role leader (L: agent,
   Members : (agent) set,
   P: symmetric_key,
   F,H: hash_func,
           Snd, Rcv: channel(dy))
     played_by L def=
   local State: nat,
         E: public_key,
         MemberContribs: (agent.nat) set,
         MemberKeys: (agent.symmetric_key) set,
         M, Mi, M1,M2,M3 : agent,
         R,Ri,R3,R1: symmetric_key,
         S,S1,SL: nat,
         Membersbis,MemberEnvoi,GroupMembers : (agent) set,
         Sall: message
   const id1: protocol_id
   init State:=0 /\ Membersbis := Members /\ MemberEnvoi := Members /\
   GroupMembers := Members /\ MemberContribs :={} /\
   MemberKeys :={}
   transition
     step1. State=0 /\ Rcv(start)
       =|> E' := new()
           /\ Snd(L.{E'}_P)
           /\ State':=1

%% debut de la phase de collecte
%% la premiere action sert à initialiser la variable SAll
     step2. State=1 /\ Rcv(M'.{R'.S'}_E)
    /\ in(M',Membersbis)
       =|> State':=2
           /\ MemberKeys':=cons(M'.R',MemberKeys)
           /\ MemberContribs':=cons(M'.S',MemberContribs)
           /\ Sall':=S'
           /\ Membersbis' := delete(M',Membersbis)

%% la boucle pour recolter les contributions des membres
     step3. State=2 /\ Rcv(M'.{R'.S'}_E)
                    /\ in(M',Membersbis)
       =|> State':=2
           /\ MemberKeys':=cons(M'.R',MemberKeys)
           /\ MemberContribs':=cons(M'.S',MemberContribs)
           /\ Sall':= Sall.S'
           /\ Membersbis' := delete(M',Membersbis)
```

```
%% Fin de la Collecte et generation de la cle de groupe Sall
     step4. State=2 /\ not(in(M2,Membersbis))
     %%le cas ou il n y a plus de membres qui n'a pas encore contribué
     %%à la clé
       =|> State':=3
           /\ SL' := new()
           /\ Sall':= Sall.SL'

%% debut de la phase d'envoie de la cle aux membres du groupe : boucle
     step5. State=3 /\ in(M3.R3,MemberKeys)
       =|> State':=3
           /\ MemberKeys' := delete(M3.R3,MemberKeys)
           /\ Snd({Sall}_R3)

%% Fin de la phase d'envoie de la cle aux membres
     step6. State=3 /\ not(in(M3.R3,MemberKeys))
       =|> State':=4

%% Debut de la phase de la récolte des messages des membres du groupe
%% cryptés par la clé de session
%% boucle : tant qu'il y a encore des participants qui n'ont pas
       %%répondu correctement
     step7. State=4 /\ Rcv(M1.{S1.H(Sall)}_F(Sall))
                    /\ in(M1.S1,MemberContribs)
       =|> State':=4
           /\ MemberContribs' := delete(M1.S1,MemberContribs)

%% Condition d'arret : on attend rien
     step8. State=4 /\ not(in(M1.S1,MemberContribs))
       =|> State':=5
           /\ GroupMembers' := cons(L,GroupMembers)
           /\ secret(F(Sall),id1,{GroupMembers'})

end role

%%The role session between the Leader and a group member
role asokan(SC, RC: channel(dy),
            L: agent,
            Members : (agent) set,
            P: symmetric_key,
            F,H: hash_func
            ) def=

   composition
leader(L,Members,P,F,H,SC,RC)
end role
```

```
%%The main role
role environment() def=
   local Members : (agent) set,
         Snd, Rcv: channel(dy)
   const m,m1,m2,m3,l: agent,
         f,h: hash_func,
         p: symmetric_key
   init Members := {m,m1,m2,m3}
   intruder_knowledge = {l,m,m1,m2,m3,p}
   composition
      asokan(Snd,Rcv,l,Members,p,f,h)
      /\ member(l,m,Members,p,f,h,Snd,Rcv)
      /\ member(l,m1,Members,p,f,h,Snd,Rcv)
      /\ member(l,m2,Members,p,f,h,Snd,Rcv)
      /\ member(l,m3,Members,p,f,h,Snd,Rcv)
end role

goal
secrecy_of id1
end goal
environment()
```

A.1.2 Instance du protocole Asokan-Ginzboorg

Instance considérée

```
%%Asokan avec deux participants
%%pptes: secret et authentication
%% First Role: a member of a group
role member (L,M: agent,
   P: symmetric_key,
   F,H: hash_func,
           Snd, Rcv: channel(dy))
     played_by M def=

   local State: nat,
         E: public_key,
         R: symmetric_key,
         S: nat,
         X,Y: nat
   const id1,ns: protocol_id
   init State:=0

   transition
step1. State=0 /\ Rcv(L.{E'}_P)
=|> R' := new()
/\ S' := new()
/\ Snd(M.{R'.S'}_E')
```

```
/\ State'=1
                /\ witness(M,L,ns,S')
step2. State=1 /\ Rcv({X'.Y'}_R)
=|> State'=3
/\ Snd(M.{S.H(X'.Y')}_F(X'.Y'))
/\ secret(F(X'.Y'),id1,{M,L})
end role

%% Second Role: The Leader of the group
role leader (L,M: agent,
   P: symmetric_key,
   F,H: hash_func,
           Snd, Rcv: channel(dy))
     played_by L def=
   local State: nat,
         E: public_key,
         R: symmetric_key,
         S,SL: nat,
         Sall: message
   const id1,id2: protocol_id
   init State:=0
   transition
step01. State=0 /\ Rcv(start)
=|> E' := new()
/\ Snd(L.{E'}_P)
/\ State'=2
step02. State=2 /\ Rcv(M.{R'.S'}_E)
=|> State'=4
/\ SL' := new()
/\ Sall':=S'.SL'
/\ Snd({Sall'}_R')
/\ secret(F(Sall'),id1,{M,L})
step03. State=4 /\ Rcv(M.{S.H(Sall)}_F(Sall))
=|> State'=5
                /\ request(L,M,ns,S)
end role

%%The role session between the Leader and a group member
role asokan(SC, RC: channel(dy),
            M, L: agent,
            P: symmetric_key,
            F,H: hash_func
            ) def=
   composition
     member(L,M,P,F,H,SC,RC)
     /\ leader(L,M,P,F,H,SC,RC)
end role
```

```
%%The main role
role environment() def=
   local Snd, Rcv: channel(dy)
   const m,l,m1,l1: agent,
         f,h: hash_func,
         p: symmetric_key
   intruder_knowledge = {l,m}
   composition
      asokan(Snd,Rcv,m,l,p,f,h)
/\ asokan(Snd,Rcv,l,m,p,f,h)
    %/\ asokan(Snd,Rcv,l1,m1,p,f,h)
    %/\ asokan(Snd,Rcv,i,l,p,f,h)
    %/\ asokan(Snd,Rcv,i,m,p,f,h)
    %/\ asokan(Snd,Rcv,l,i,p,f,h)
end role

goal
secrecy_of id1
authentication_on ns
end goal
environment()
```

Résumé de l'attaque pour cette spécification

```
SUMMARY
  UNSAFE

DETAILS
  ATTACK_FOUND
  TYPED_MODEL

PROTOCOL
  /home/avispa/web-interface-computation/./tempdir/workfileQu8339.if

GOAL
  Authentication attack on (l,m,ns,n11(S))

BACKEND
  CL-AtSe

STATISTICS

  Analysed   : 49 states
  Reachable  : 21 states
  Translation: 0.01 seconds
  Computation: 0.00 seconds
```

```
ATTACK TRACE
 i -> (m,7):  start
 (m,7) -> i:  m.{n15(E)}_p

 i -> (l,4):  start
 (l,4) -> i:  l.{n5(E)}_p

 i -> (l,6):  m.{n5(E)}_p
 (l,6) -> i:  l.{n11(R).n11(S)}_n5(E)
              & Witness(l,m,ns,n11(S));

 i -> (l,4):  m.{n11(R).n11(S)}_n5(E)
 (l,4) -> i:  {n11(S).n6(SL)}_n11(R)
              & Secret({n11(S).n6(SL)}_f,set_90);  Add m to set_90;  Add l to set_90;

 i -> (m,3):  l.{n15(E)}_p
 (m,3) -> i:  m.{n1(R).n1(S)}_n15(E)
              & Witness(m,l,ns,n1(S));

 i -> (m,7):  l.{n1(R).n1(S)}_n15(E)
 (m,7) -> i:  {n1(S).n16(SL)}_n1(R)
              & Secret({n1(S).n16(SL)}_f,set_99);  Add l to set_99;  Add m to set_99;

 i -> (l,6):  {n11(S).n6(SL)}_n11(R)
 (l,6) -> i:  l.{n11(S).{n11(S).n6(SL)}_h}_({n11(S).n6(SL)}_f)
              & Secret({n11(S).n6(SL)}_f,set_97);  Add l to set_97;  Add m to set_97;

 i -> (l,4):  m.{n11(S).{n11(S).n6(SL)}_h}_({n11(S).n6(SL)}_f)
 (l,4) -> i:  ()
              & Request(l,m,ns,n11(S));
```

A.2 Spécification en HLPSL du protocole GDH

A.2.1 Première spécification

spécification en HLPSL

```
role initiator (M1: agent,
   MemberGroup: (agent) set,
           Alpha: nat,
           Snd, Rcv: channel(dy))
     played_by M1 def=
   local State: nat,
         Xx: message,
         Key: symmetric_key,
         R1,X: nat
```

```
   const id: protocol_id
   init State:=0
   transition
%
    step1. State=0 /\ Rcv(start)
       =|>  R1' := new()
           /\ Snd(exp(Alpha,1).exp(Alpha,R1'))
           /\ State':=1
%
%
    step2. State=1 /\ Rcv(exp(Xx',X'))
       =|> State':=2
           /\ Key' := exp(exp(Xx',X'),R1)
           /\ secret(Key',id,{MemberGroup})
%
end role

role medium (Mi: agent,
   MemberGroup: (agent)set,
           Alpha: nat,
           Snd, Rcv: channel(dy))
     played_by Mi def=
   local State: nat,
         SndMsg,MsgToTreat: message,
         X1,Xi,X2,Xexp1,Xexpi1, Xexpf,Xexp11,Xexpi11:message,
         Key: symmetric_key,
         R2,R1,Ri,Ri1,Ri11,R11: nat
   const id: protocol_id
   init State:=0
   transition
%
    step2. State=0 /\ Rcv(exp(Xexp1',R1').exp(Xexp11',R11').X2')
       =|>  R2' := new()
           /\ MsgToTreat':= exp(Xexp11',R11').X2'
           /\ SndMsg':= exp(exp(Xexp1',R1'),R2')
           /\ State':=1
%
    step3. State=1 /\ MsgToTreat = exp(Xexpi1',Ri1')
       =|> SndMsg' := SndMsg.exp(Xexpi1',Ri1').exp(exp(Xexpi1',Ri1'),R2)
           /\ Snd(SndMsg')
           /\ State':=2
%
    step4. State=1 /\ MsgToTreat = exp(Xexpi1',Ri1').exp(Xexpi11',Ri11').Xi'
       =|> SndMsg' := SndMsg.exp(exp(Xexpi1',Ri1'),R2)
           /\   MsgToTreat':= exp(Xexpi11',Ri11').Xi'
           /\ State':=1
%
    step5. State=2 /\ Rcv(exp(Xexpf',R1'))
```

```
       =|> State':=3
           /\ Key' := exp(exp(Xexpf,R1'),R2)
           /\ secret(Key',id,{MemberGroup})
%
end role

role last (Ml: agent,
   MemberGroup: (agent) set,
           Alpha: nat,
%           Pk1,Pk2: public_key,
           Snd, Rcv: channel(dy))
     played_by Ml def=
   local State: nat,
         X2,Xi,SndMsg,Xexp1,Xexp11,MsgToTreat: message,
         Key: symmetric_key,
         R3,R1,R11: nat
   const id: protocol_id
   init State:=0
   transition
%
    step2. State=0 /\ Rcv(exp(Xexp1',R1').X2')
       =|>  R3' := new()
           /\ MsgToTreat':= exp(Xexp1',R1').X2'
%           /\ Snd(exp(exp(Xexp1',R1'),R3'))
           /\ State':=1
%
    step3. State=1 /\ MsgToTreat = exp(Xexp1',R1')
       =|>  Key' := exp(exp(Alpha,R1'),R3)
           /\ secret(Key',id,{MemberGroup})
           /\ State':=2
%
    step4. State=1 /\ MsgToTreat = exp(Xexp1',R1').exp(Xexp11',R11').Xi'
       =|>  Snd(exp(exp(Xexp1',R1'),R3))
           /\ MsgToTreat':= exp(Xexp11',R11').Xi'
           /\ State':=1
end role

role gdh(SC, RC: channel(dy),
            M1,Ml,Mi: agent,
    MemberGroup : (agent) set,
            Alpha: nat
            ) def=

   composition
 initiator(M1,MemberGroup,Alpha,SC,RC)
 /\ medium(Mi,MemberGroup,Alpha,SC,RC)
 /\ last(Ml,MemberGroup,Alpha,SC,RC)
end role
```

```
role environment() def=
   local MemberGroup: (agent) set,
         Snd, Rcv: channel(dy)
   const m1,mi,ml: agent,
         alpha: nat
   init MemberGroup := {m1,mi,ml}
   intruder_knowledge = {m1,mi,ml}
   composition
   gdh(Snd, Rcv,m1,ml,mi,MemberGroup,alpha)
end role

goal
secrecy_of id
%authentication_on n1,n2,n3,n4
end goal

environment()
```

Attaque trouvée par OFMC

```
% OFMC
% Version of 2006/02/13
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
PROTOCOL
  /home/avispa/web-interface-computation/./tempdir/workfileTu3962.if
GOAL
  secrecy_of_id
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.50s
  visitedNodes: 120 nodes
  depth: 3 plies
ATTACK TRACE
i -> (m1,3): start
(m1,3) -> i: exp(alpha,1).exp(alpha,R1(1))
i -> (ml,3): exp(exp(alpha,R1(1)),x247).exp(alpha,R1(1)).x261
(ml,3) -> i: exp(exp(exp(alpha,R1(1)),x247),R3(2))
i -> (m1,3): exp(alpha,R3(2))
i -> (i,17): exp(exp(alpha,R3(2)),R1(1))
i -> (i,17): exp(exp(alpha,R3(2)),R1(1))
```

A.2.2 Deuxième spécification

Spécification

```
role initiator (M1: agent,
MemberGroup : (agent) set,
Alpha: nat,
Snd, Rcv: channel(dy))
played_by M1 def=
   local State: nat,
         X1,X2: message,
         Key: symmetric_key,
         X,R1: nat
   const id: protocol_id
   init State:=0
   transition
%
    step1. State=0 /\ Rcv(start)
       =|>  R1' := new()
           /\ Snd(exp(Alpha,1).exp(Alpha,R1'))
           /\ State':=1
%
    step2. State=1 /\ Rcv(exp(X2',X').X1')
       =|> State':=2
           /\ Key' := exp(exp(X2',X'),R1)
           /\ secret(Key',id,{MemberGroup})
%
end role

role medium     (Mi: agent,
MemberGroup : (agent) set,
Pos : nat,
Alpha: nat,
Snd, Rcv: channel(dy))
played_by Mi def=
   local State: nat,
         SndMsg,MsgToTreat,MsgNeeded : message,
         X,Xx,Xxx,X1,Xi,X2,Xexp1,Xexpi1,Xexpi11,Xexp11, Xexpf:message,
         Key: symmetric_key,
         R,R2,R1,Ri,Ri1,CurrentPos,R11,Ri11: nat
   const id: protocol_id
   init State:=0
   transition
%% En recevant un message du membre précédant,
%% l'intermediaire génère le prochain message par récursion
% Debut premiere etape
    step2. State=0 /\ Rcv(exp(Xexp1',R1').exp(Xexp11',R11').X2')
       =|>  R2' := new()
           /\ MsgToTreat':= exp(Xexp11',R11').X2'
```

```
           /\ SndMsg':= exp(exp(Xexp1',R1'),R2')
           /\ State':=1
%
    step3. State=1 /\ MsgToTreat = exp(Xexpi1',Ri1')
       =|> SndMsg' := SndMsg.exp(Xexpi1',Ri1').exp(exp(Xexpi1',Ri1'),R2)
           /\ Snd(SndMsg')
           /\ State':=2
%
    step4. State=1 /\ MsgToTreat = exp(Xexpi1',Ri1').exp(Xexpi11',Ri11').Xi'
       =|> SndMsg' := SndMsg.exp(exp(Xexpi1',Ri1'),R2)
           /\   MsgToTreat':= exp(Xexpi11',Ri11').Xi'
% Fin premiere etape

%%Pour calculer sa cle de groupe,
%% le membre utilise sa connaissance
%%de sa position dans le groupe
    step5. State=2 /\ not(Pos=1) /\ Rcv(exp(Xexpf',R1').X')
       =|> State':=3
   /\ CurrentPos' := 2
   /\ MsgNeeded' := X'
%
    step6. State=3 /\ CurrentPos=Pos /\ MsgNeeded=(exp(Xxx',R').Xx')
       =|> State':=4
           /\ Key' := exp(exp(Xxx',R'),R2)
           /\ secret(Key',id,{MemberGroup})
%
    step7. State=3 /\ not(CurrentPos=Pos) /\ MsgNeeded=(exp(Xxx',R').Xx')
       =|> State':=3
   /\ CurrentPos' := CurrentPos+1
   /\ MsgNeeded' := Xx'
end role

role last       (Ml: agent,
MemberGroup : (agent) set,
Alpha: nat,
Snd, Rcv: channel(dy))
played_by Ml def=
   local State: nat,
         X2,Xi,SndMsg,Xexp1,MsgToTreat,Xexp11,Xexpi1,Xexpi: message,
         Key: symmetric_key,
         R3,R1,R11,Ri,Ri1: nat
   const id: protocol_id
   init State:=0
   transition
%
    step2. State=0 /\ Rcv(exp(Xexp1',R1').exp(Xexp11',R11').X2')
       =|>  R3' := new()
           /\ MsgToTreat':= exp(Xexp11',R11').X2'
```

```
           /\ SndMsg' := exp(exp(Xexp1',R1'),R3')
           /\ State':=1
%
    step3. State=1 /\ MsgToTreat = exp(Xexp1',R1')
       =|>  Snd(SndMsg)
           /\ Key' := exp(exp(Alpha,R1'),R3)
           /\ secret(Key',id,{MemberGroup})
           /\ State':=2
%
    step4. State=1 /\ MsgToTreat = exp(Xexpi',Ri').exp(Xexpi1',Ri1').Xi'
       =|>  SndMsg' := SndMsg.exp(exp(Xexpi',Ri'),R3)
           /\ MsgToTreat':= exp(Xexpi1',Ri1').Xi'
           /\ State':=1
end role

role gdh(SC, RC: channel(dy),
            M1,Ml: agent,
    MemberGroup : (agent) set,
            Alpha: nat
            ) def=

   composition
 initiator(M1,MemberGroup,Alpha,SC,RC)
 /\ last(Ml,MemberGroup,Alpha,SC,RC)
end role

role environment() def=
   local MemberGroup: (agent) set,
         Snd, Rcv: channel(dy)
   const m1,mi,ml: agent,
         alpha: nat
   init MemberGroup := {m1,mi,ml}
   intruder_knowledge = {m1,mi,ml,alpha}
   composition
   gdh(Snd, Rcv,m1,ml,MemberGroup,alpha)
/\ medium(mi,MemberGroup,2,alpha,Snd,Rcv)
end role

goal
secrecy_of id
end goal
environment()
```

Attaque

```
SUMMARY
  UNSAFE
```

```
DETAILS
  ATTACK_FOUND
  TYPED_MODEL
  BOUNDED_SEARCH_DEPTH
PROTOCOL
  /users/cassis/chridi/Thesis/TestHLPSL/agdhv2/update/gdh4-1.if
GOAL
  Secrecy attack on (exp(alpha,n5(R3)*R1(6)))
BACKEND
  CL-AtSe
STATISTICS
  Analysed   : 2 states
  Reachable  : 1 states
  Translation: 0.03 seconds
  Computation: 0.00 seconds
ATTACK TRACE
 i -> (m1,3):  start
 (m1,3) -> i:  exp(alpha,1).exp(alpha,n1(R1))
 i -> (ml,4):  exp(alpha,R1(6)).exp(Xexp1(6),R1(6))
 (ml,4) -> i:  ()
 i -> (ml,4):  ()
 (ml,4) -> i:  exp(alpha,R1(6)*n5(R3))
               & Secret(exp(alpha,n5(R3)*R1(6)),set_108);
               & Add set_94 to set_108;
```

A.3 Spécification en HLPSL du protocole Tanaka-Sato

A.3.1 Première spécification

```
%% the tanaka Sato protocol
%% with two roles: a leader and a normal member
%% First Role: a member of a group
role member (L,M: agent,
           Km: symmetric_key,
           Snd, Rcv: channel(dy))
     played_by M def=

   local State: nat,
         M2,M1 : message,
         Im,Gk,KG: symmetric_key,
         CGk,CGKm: symmetric_key
   const id1,id2,id3: protocol_id
   init State:=0
   transition
%% le cas de Join
     step11. State=0 /\ Rcv(start)
       =|>  Snd({join}_Km)
           /\ State':=1
```

```
     step21. State=1 /\ Rcv({Im'.Gk'}_Km)
       =|> State':=2
           /\ CGKm':=Gk'
           /\ secret(Gk',id1,{L,M})
%% apres avoir fait un join (etre en etat 2), le membre peut faire
%% une des requetes suivantes : read,send,leave
%% le cas de read : il retourne a l etat 2
     step14. State=2 /\ Rcv({M2'}_KG')
       =|> Snd({read.CGKm}_Im)
          /\ State' :=7
     step23. State=7 /\ Rcv({CGk'}_Im)
       =|> CGKm':=CGk'
           /\ State':=2
           /\ secret(CGk',id4,{L,M})
%% le cas de send : il retourne a l etat 2
     step13. State=2 /\ Rcv(start)
       =|> Snd({send.CGKm}_Im)
          /\ State' :=5
     step23. State=5 /\ Rcv({CGk'}_Im)
       =|> M1':= new()
           /\ Snd({M1'}_CGk')
           /\ CGKm':=CGk'
           /\ State':=2
           /\ secret(CGk,id2,{L,M})
%% le cas de leave
     step12. State=2 /\ Rcv(start)
       =|> Snd({leave}_Im)
          /\ State' :=3
     step22. State=3 /\ Rcv({ackleave}_Im)
       =|> State':=4
end role

%% Second Role: The Leader of the group
role leader (L: agent,
           LongTermKey: (agent.symmetric_key) set,
           Snd, Rcv: channel(dy))
     played_by L def=
   local State: nat,
         GroupMembers: (agent) set,
         GroupIndivMembers: (agent.symmetric_key) set,
         M: agent,
         Im,Gk,CGk,GKm,Km,Ikm: symmetric_key
   const id1,id2,id3,id4: protocol_id
   init State:=0 /\ GroupMembers := {L} /\ GroupIndivMembers := {}
   transition
%% Join
     step11. State=0 /\ Rcv({join}_Km')
```

```
                    /\ in(M'.Km',LongTermKey)
                    /\ not(in(M',GroupMembers))
=|> Im' := new()
           /\ Gk' := new()
           /\ CGk' := Gk'
           /\ Snd({Im'.Gk'}_Km')
           /\ GroupMembers' := cons(M',GroupMembers)
           /\ GroupIndivMembers' := cons(M'.Im',GroupIndivMembers)
           /\ State':=0
           /\ secret(Gk',id1,GroupMembers)
%% leave
     step12. State=0 /\ Rcv({leave}_Ikm')
                     /\ in(M'.Ikm',GroupIndivMembers)
       =|>  Snd({ackleave}_Ikm')
            /\ CGk' := new()
            /\ GroupMembers' := delete(M',GroupMembers)
            /\ GroupIndivMembers' := delete(M'.Ikm',GroupIndivMembers)
            /\ State':=0
%%send
     step13. State=0 /\ Rcv({send.GKm'}_Ikm')
                     /\ in(M'.Ikm',GroupIndivMembers)
       =|> Snd({CGk}_Ikm')
           /\ State':=0
           /\ secret(CGk,id2,GroupMembers)
%%read
     step14. State=1 /\ Rcv({read.GKm'}_Ikm')
                     /\ in(M'.Ikm',GroupIndivMembers)
       =|> Snd({CGk}_Ikm')
           /\ State':=0
           /\ secret(CGk,id4,GroupMembers)
end role

%%The role session between the Leader and a group member
role tanaka(SC, RC: channel(dy),
            L: agent,
            LongTermKey: (agent.symmetric_key)set
            ) def=
   composition
      leader(L,LongTermKey,SC,RC)
end role

%%The main role
role environment() def=
   local LongKey: (agent.symmetric_key)set,
         Snd, Rcv: channel(dy)
   const m1,m2,m3,l: agent,
         leave,join,send,ackleave,mess,read: text,
         km1,km2,km3,ki: symmetric_key
```

```
   init LongKey := {m1.km1, m2.km2, m3.km3, i.ki}
   intruder_knowledge = {l,m1,m2,m3,ki}
   composition
      tanaka(Snd,Rcv,l,LongKey)
      /\ member(l,m1,km1,Snd,Rcv)
      /\ member(l,m2,km2,Snd,Rcv)
      /\ member(l,m3,km3,Snd,Rcv)
      /\ member(l,i,ki,Snd,Rcv)
end role

goal
secrecy_of id1,id2,id3,id4
end goal

environment()
```

A.3.2 Deuxième spécification

```
%%un agent qui n appartient pas au groupe
role nonmember (L,NM: agent,
           Km: symmetric_key,
           Snd, Rcv: channel(dy))
     played_by NM def=
   local State: nat,
         Im,Gk,CGKm: symmetric_key
   const id1: protocol_id
   init State:=0
   transition
%% le cas de Join
     step11. State=0 /\ Rcv(start)
       =|>  Snd({join}_Km)
           /\ State':=1
     step21. State=1 /\ Rcv({Im'.Gk'}_Km)
       =|> State':=2
           /\ CGKm':=Gk'
           /\ secret(Gk',id1,{L,NM})
end role
%% un membre qui appartient au groupe
role member (L,M: agent,
           Km: symmetric_key,
           GroupK: symmetric_key, %% la cle du groupe
           Snd, Rcv: channel(dy))
     played_by M def=
   local State: nat,
         M2,M1 : message,
         Im,Gk,KG: symmetric_key,
         CGk,CGKm: symmetric_key
   const id1,id2,id3: protocol_id
```

```
   init State:=2 /\ CGKm := GroupK
   transition
%% apres avoir fait un join (etre en etat 2), le membre peut faire
%% une des requetes suivantes : read,send,leave
%% le cas de read : il retourne a l etat 2
     step14. State=2 /\ Rcv({M2'}_KG')
       =|> Snd({read.CGKm}_Im)
          /\ State' :=7
     step23. State=7 /\ Rcv({CGk'}_Im)
       =|> CGKm':=CGk'
           /\ State':=2
           /\ secret(CGk',id4,{L,M})
%% le cas de send : il retourne a l etat 2
     step13. State=2 /\ Rcv(start)
       =|> Snd({send.CGKm}_Im)
          /\ State' :=5
     step23. State=5 /\ Rcv({CGk'}_Im)
       =|> M1':= new()
           /\ Snd({M1'}_CGk')
           /\ CGKm':=CGk'
           /\ State':=2
           /\ secret(CGk,id2,{L,M})
%% le cas de leave
     step12. State=2 /\ Rcv(start)
       =|> Snd({leave}_Im)
          /\ State' :=3
     step22. State=3 /\ Rcv({ackleave}_Im)
       =|> State':=4
end role

%% Second Role: The Leader of the group
role leader (L: agent,
           LongTermKey: (agent.symmetric_key) set,
           Snd, Rcv: channel(dy))
     played_by L def=
   local State: nat,
         GroupMembers: (agent) set,
         GroupIndivMembers: (agent.symmetric_key) set,
         M: agent,
         Im,Gk,CGk,GKm,Km,Ikm: symmetric_key
   const id1,id2,id3,id4: protocol_id
   init State:=0 /\ GroupMembers := {L} /\ GroupIndivMembers := {}
   transition
%% Join
     step11. State=0 /\ Rcv({join}_Km')
                     /\ in(M'.Km',LongTermKey)
                     /\ not(in(M',GroupMembers))
=|> Im' := new()
```

```
           /\ Gk' := new()
           /\ CGk' := Gk'
           /\ Snd({Im'.Gk'}_Km')
           /\ GroupMembers' := cons(M',GroupMembers)
           /\ GroupIndivMembers' := cons(M'.Im',GroupIndivMembers)
           /\ State':=0
           /\ secret(Gk',id1,GroupMembers)
%% leave
     step12. State=0 /\ Rcv({leave}_Ikm')
                     /\ in(M'.Ikm',GroupIndivMembers)
       =|>  Snd({ackleave}_Ikm')
            /\ CGk' := new()
            /\ GroupMembers' := delete(M',GroupMembers)
            /\ GroupIndivMembers' := delete(M'.Ikm',GroupIndivMembers)
            /\ State':=0
%%send
     step13. State=0 /\ Rcv({send.GKm'}_Ikm')
                     /\ in(M'.Ikm',GroupIndivMembers)
       =|> Snd({CGk}_Ikm')
           /\ State':=0
           /\ secret(CGk,id2,GroupMembers)
%%read
     step14. State=1 /\ Rcv({read.GKm'}_Ikm')
                     /\ in(M'.Ikm',GroupIndivMembers)
       =|> Snd({CGk}_Ikm')
           /\ State':=0
           /\ secret(CGk,id4,GroupMembers)
end role

%%The role session between the Leader and a group member
role tanaka(SC, RC: channel(dy),
            L: agent,
            LongTermKey: (agent.symmetric_key)set
            ) def=
   composition
      leader(L,LongTermKey,SC,RC)
end role

%%The main role
role environment() def=
   local LongKey: (agent.symmetric_key)set,
         Snd, Rcv: channel(dy)
   const m1,m2,m3,l: agent,
         leave,join,send,ackleave,mess,read: text,
         km1: symmetric_key
   init LongKey := {m1.km1}
   intruder_knowledge = {l,m1}
   composition
```

```
      tanaka(Snd,Rcv,l,LongKey)
      /\ nonmember(l,m1,km1,Snd,Rcv)
end role

goal
secrecy_of id1,id2,id3,id4
end goal

environment()
```

A.4 Spécification en HLPSL du protocole Iolus

```
%% the iolus  protocol
%% with two roles: a leader and a normal member
%% First Role: a member of a group
role member (L,M: agent,
           Km: symmetric_key,
           Snd, Rcv: channel(dy))
     played_by M def=
   local State: nat,
         Im,Gk: symmetric_key,
         CGk,CGKm: symmetric_key
   const id1,id: protocol_id
   init State:=0
   transition
% JOIN  %%%%%%%%%%
    step1. State=0 /\ Rcv(start)
       =|>  Snd({join}_Km)
           /\ State':=1
%
     step2. State=1 /\ Rcv({Im'.Gk'}_Km)
       =|> State':=2
           /\ CGKm':=Gk'
     step10. State=2 /\ Rcv({CGk'}_CGKm)
       =|> State':= 6
           /\ CGKm':=CGk'
           /\ secret(CGKm',id,{L,M})
%% Fin JOIN %%%%%%%%%%%%%%%%%%
%% Leave
     step12. State=6 /\ Rcv(start)
       =|> Snd({leave}_Im)
          /\ State' :=3
%%%fin leave
     step13. State=6 /\ Rcv(start)
       =|> Snd({mess}_CGKm)
          /\ State' :=6
     step100. State=6 /\ Rcv({CGk'}_Im)
```

```
       =|> State':= 6
           /\ CGKm':=CGk'
           /\ secret(CGKm',id1,{L,M})
end role

%% Second Role: The Leader of the group
role leader (L: agent,
             LongTermKey: (agent.symmetric_key) set,
             Snd, Rcv: channel(dy))
     played_by L def=
   local State: nat,
         GroupMembers: (agent) set,
         GroupIndivMembers,GpInMembers: (agent.symmetric_key) set,
         M, Mi,M2,M3: agent,
         Im,Gk,CGk,GKm,Km,IKm,Ikmi: symmetric_key
   const id1,id2: protocol_id
   init State:=0 /\ GroupMembers := {L} /\ GroupIndivMembers := {} /\ GpInMembers :={}
   transition
     step0. State=0 /\ Rcv(start)
       =|> CGk' := new()
           /\ State':=2
     step1. State=2  /\ Rcv({join}_Km')
                     /\ in(M'.Km',LongTermKey)
                     /\ not(in(M',GroupMembers))
       =|> Im' := new()
           /\ Gk' := new()
           /\ Snd({Gk'}_CGk)
           /\ CGk' := Gk'
           /\ Snd({Im'.Gk'}_Km')
           /\ GroupMembers' := cons(M',GroupMembers)
           /\ GroupIndivMembers' := cons(M'.Im',GroupIndivMembers)
           /\ State':=2
           /\ secret(CGk',id,GroupMembers')
     step12. State=2 /\ Rcv({leave}_IKm')
                     /\ in(M2'.IKm',GroupIndivMembers)
       =|>  CGk' := new()
            /\ GroupMembers' := delete(M2',GroupMembers)
    /\ GroupIndivMembers' := delete(M2'.IKm',GroupIndivMembers)
    /\ GpInMembers':=GroupIndivMembers'
           /\ State':=3
           /\ secret(CGk',id1,GroupMembers')
     step12. State=3 /\ in (Mi'.Ikmi',GpInMembers)
       =|>  Snd({CGk}_Ikmi')
    /\ GpInMembers':=delete(Mi'.Ikmi',GpInMembers)
           /\ State':=3
     step12. State=3 /\ not(in (Mi'.Ikmi',GroupIndivMembers))
       =|>   State':=2
end role
```

```
%%The role session between the Leader and a group member
role tanaka(SC, RC: channel(dy),
            L: agent,
            LongTermKeys : (agent.symmetric_key) set
            ) def=
   composition
leader(L,LongTermKeys,SC,RC)
end role

%%The main role
role environment() def=
   local Snd, Rcv: channel(dy),
         LongKey : (agent.symmetric_key) set
   const m,l,m1,m2,m3,m4,m5: agent,
         leave,join,send,ackleave,mess,read: text,
         km,ki,km1,km2,km3,km4,km5: symmetric_key
   init LongKey := {m.km}
   intruder_knowledge = {l,m}
   composition
      tanaka(Snd,Rcv,l,LongKey)
      /\ member(l,m,km,Snd,Rcv)
end role
goal
secrecy_of id1,id
end goal
environment()
```

Résumé

Les protocoles cryptographiques sont cruciaux pour sécuriser les transactions électroniques. La confiance en ces protocoles peut être augmentée par l'analyse formelle de leurs propriétés de sécurité. Bien que beaucoup de travaux aient été dédiés pour les protocoles classiques comme le protocole de Needham-Schroeder, très peu de travaux s'adressent à la classe des protocoles de groupe dont les caractéristiques principales sont : les propriétés de sécurité spécifiques qu'ils doivent satisfaire, et le nombre arbitraire des participants qu'ils impliquent.

Cette thèse comprend deux contributions principales. La première traite la première caractéristique des protocoles de groupe. Pour cela, nous avons défini un modèle appelé *modèle de services* que nous avons utilisé pour proposer une stratégie de recherche d'attaques se basant sur la résolution de contraintes. L'approche proposée a permis de retrouver d'anciennes attaques et d'en découvrir de nouvelles sur quelques protocoles de groupe. Certaines attaques ont aussi pu être généralisées pour couvrir le cas de n participants. La deuxième contribution principale de cette thèse consiste à définir un modèle *synchrone* qui généralise les modèles standards de protocoles en permettant les listes non bornées à l'intérieur des messages. Ceci est assuré par l'introduction d'un nouvel opérateur appelé *mpair* qui représente une liste construite sur un même patron. Dans ce modèle étendu, nous avons proposé une procédure de décision pour une classe particulière des protocoles de groupe appelée classe de *protocoles bien-tagués* avec *clefs autonomes*, en présence d'un intrus *actif* et avec des clefs *composées*.

Mots-clés: protocole de groupe, listes paramétrées, protocole bien-tagué, propriété de sécurité, attaque, décidabilité, clefs autonomes, modèle synchrone, modèle de services.

Abstract

Cryptographic protocols are crucial for securing electronic transactions. The confidence in these protocols can be increased by the formal analysis of their security properties. Although many works have been dedicated to standard protocols like Needham-Schroeder, very few address the class of group protocols whose main characteristics are : the specific security properties that they must satisfy, and the arbitrary number of participants they imply.

This thesis provides two main contributions. The first one deals with the first characteristic of group protocols. For that, we defined a model called the *services model* which we used to propose a strategy for flaws detection based on constraints solving. The suggested approach allows us to find known attacks and new ones on some group protocols. Some attacks have been also generalized to cover the case of n participants. The second main contribution of this thesis consists in defining a *synchronous* model, that generalizes standard protocol models by permitting unbounded lists inside messages. This is ensured by the introduction of a new operator called *mpair* which represents a list built on the same pattern. In this extended model, we have proposed a decision procedure for a particular class of group protocols called the class of *well-tagged protocols* with *autonomous keys*, in presence of an *active* intruder and with *composed* keys.

Keywords: group protocol, parameterized lists, well-tagged protocol, security property, attack, decidability, autonomous keys, synchronous model, services model.